Biotic Prediction

Building the Computational Technology Infrastructure
for Public Health and Environmental Forecasting

Software Design Document

BP-SDD-1.1

Task Agreement: GSFC-CT-1

August 22, 2003

Contents

List of Tables

List of Figures

1 Introduction

1.1 Invasive Species Forecasting System

This project is developing the high-performance, computational technology infrastructure needed to analyze the past, present, and future geospatial distributions of living components of Earth environments. This involves moving a suite of key predictive, geostatistical biological models into a scalable, cost-effective cluster computing framework; collecting and integrating diverse Earth observational datasets for input into these models; and deploying this functionality as a Web-based service. The resulting infrastructure will be used in the ecological analysis and prediction of exotic species invasions. This new capability will be deployed at the USGS Fort Collins Science Center and extended to other scientific communities through the USGS National Biological Information Infrastructure program.

1.2 Software Design Document Overview

This Software Design Document has been prepared in accordance with NASA/GSFC's "Recommended Approach to Software Development Revision 3". The sections included in this document are as follows:

Section 1. Introduction
1.1 Invasive Species Forecasting System — Abstract describing the project.
1.2 Software Design Document — List describing sections of this SDD.

Section 2. Design Overview
2.1 Design Drivers — The design drivers for this project and their order of importance.
2.2 Reuse Strategy — Statement regarding reuse strategy of the system.
2.3 System Design — Discussion and high-level diagrams of system design and interfaces.
2.4 Design Status — Constraints, assumptions, and TBD requirements.
2.5 Development Environment — A description of the current development environment.

Section 3. Operations Overview
3.1 Operations Scenarios — Operations scenarios and scripts.
3.2 System Performance — System performance and considerations.

Section 4. Design Description
4.1 Subsystem Capability — Overall subsystem capability.
4.2 Processing Restrictions — Assumptions and restrictions to processing.
4.3 Subsystem — Discussion and high-level diagrams of subsystem.
4.4 Input/Output — High-level description of input and output.
4.5 Processing — Detailed description of processing.
4.6 Structure — Structure charts expanded to the object-oriented level.
4.7 Data Storage — Internal Storage requirements.
4.8 Graphical User Interface — The project's interactive graphical system.

Section 5. Data Interfaces
5.1 Database Tables — Descriptions of database tables used.
5.2 Entity Relationship (ER) Diagram — ER Diagram.

Appendix A — Glossary
Appendix B — Structure Charts & Object Oriented Design
Appendix C — ISFS Developer Setup/Deployment Instructions

2 Design Overview

2.1 Design Drivers

The Invasive Species Forecasting System is one of NASA's Computational Technologies (CT) projects. The CT project objectives (http://ct.gsfc.nasa.gov/overview.html) include the following:

> "The goal of NASA's Computational Technologies (CT) Project is to demonstrate the potential afforded by teraFLOPS (trillion floating-point operations per second) performance to further our understanding and ability to predict the dynamic interaction of physical, chemical, and biological processes affecting the solar-terrestrial environment and the universe."

The ISFS project is driven by five specific CT objectives:

- Support the development of massively parallel, scalable, multidisciplinary models and data processing algorithms.
- Make available prototype, scalable, parallel architectures and massive data storage systems to CT researchers.
- Prepare the software environments to facilitate scientific exploration and sharing of information and tools.
- Develop data management tools for high-speed access management and visualization of data with teraFLOPS computers.
- Demonstrate the scientific and computational impact for Earth and space science applications.

2.2 Reuse Strategy

The ISFS software is being prepared and documented so that it will be available for reuse. This includes documentation and packaging of the project software and hardware specifications for cluster computing. Please refer to Appendix C, "ISFS Developer Setup/Deployment Instructions" as guideline to installing and running the application.

2.3 System Design

The ISFS system will have users from government agencies, universities, industry, and the public. To the users, the ISFS is deployed as a web browser accessible system that will present options for applying a series of models to available datasets yielding predictive result sets. The system can ingest data from different sources and in different formats and existing models can be either run, or new ones created. The system outputs maps and additional information depicting the applied model and the predicted species distributions.

2.3.1 External Interfaces

The primary means of interface to the ISFS will be a W3 standards compliant web browser. All GUI development efforts for external interfaces will be through web-based client architecture that uses HTTP as the primary means for interacting with the system. Supplementary interfaces for ingest may include a secured FTP push/pull technique that will be scheduled through the GUI.

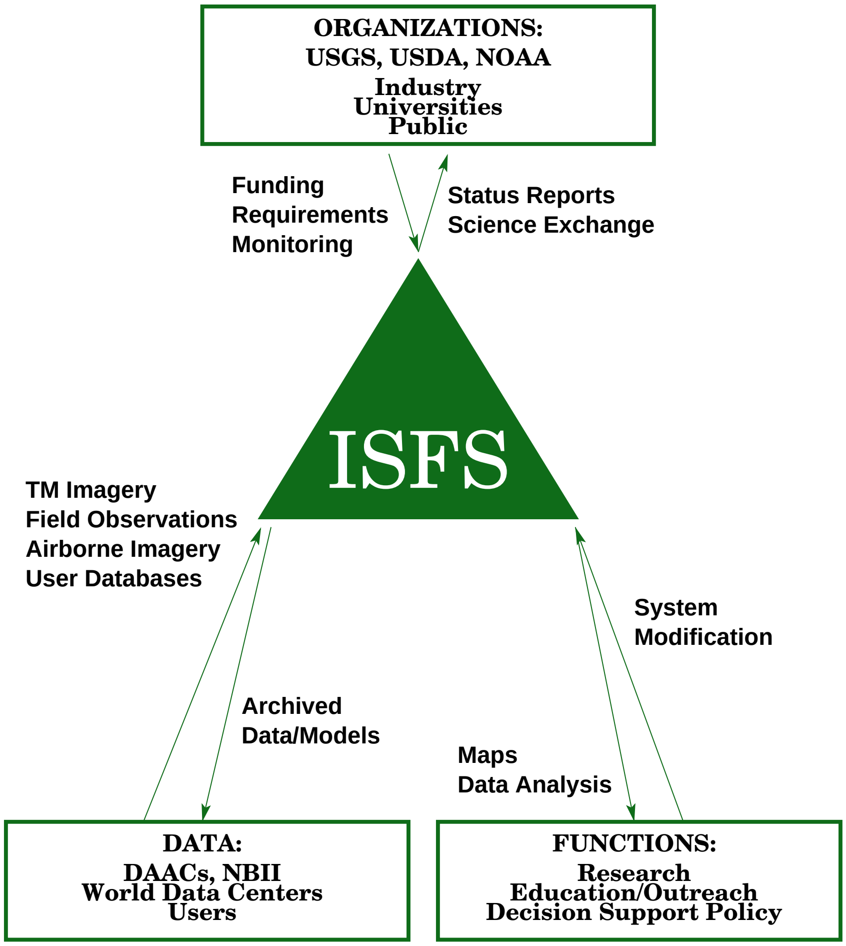


Figure 1. Context Diagram

Standing orders for required data products will be accomplished through subscriptions to the varied data sources including the DAAC, NBII, USGS, and other data sources TBD. Different levels of access for public, model user, model builder, and developer will be established and documented in the SRD.

The user interface is implemented through an HTTP connection to browser-based GUI to select models and datasets, and to apply the selected model to the selected dataset. The GUI is also the vector for returning processed output to the end user.

2.3.2 Internal Architecture

The overall internal architecture of the Invasive Species Forecasting System is presented in Figure 2. It consists of three major conceptual layers:

1. A Front End Layer that provides a suite of graphical user interfaces to the various components of the system that must be accessed by the various classes of users of the system,

2. An Application Layer whose subsystems support the activities, computations, and workflows that define the ISFS modeling process, and

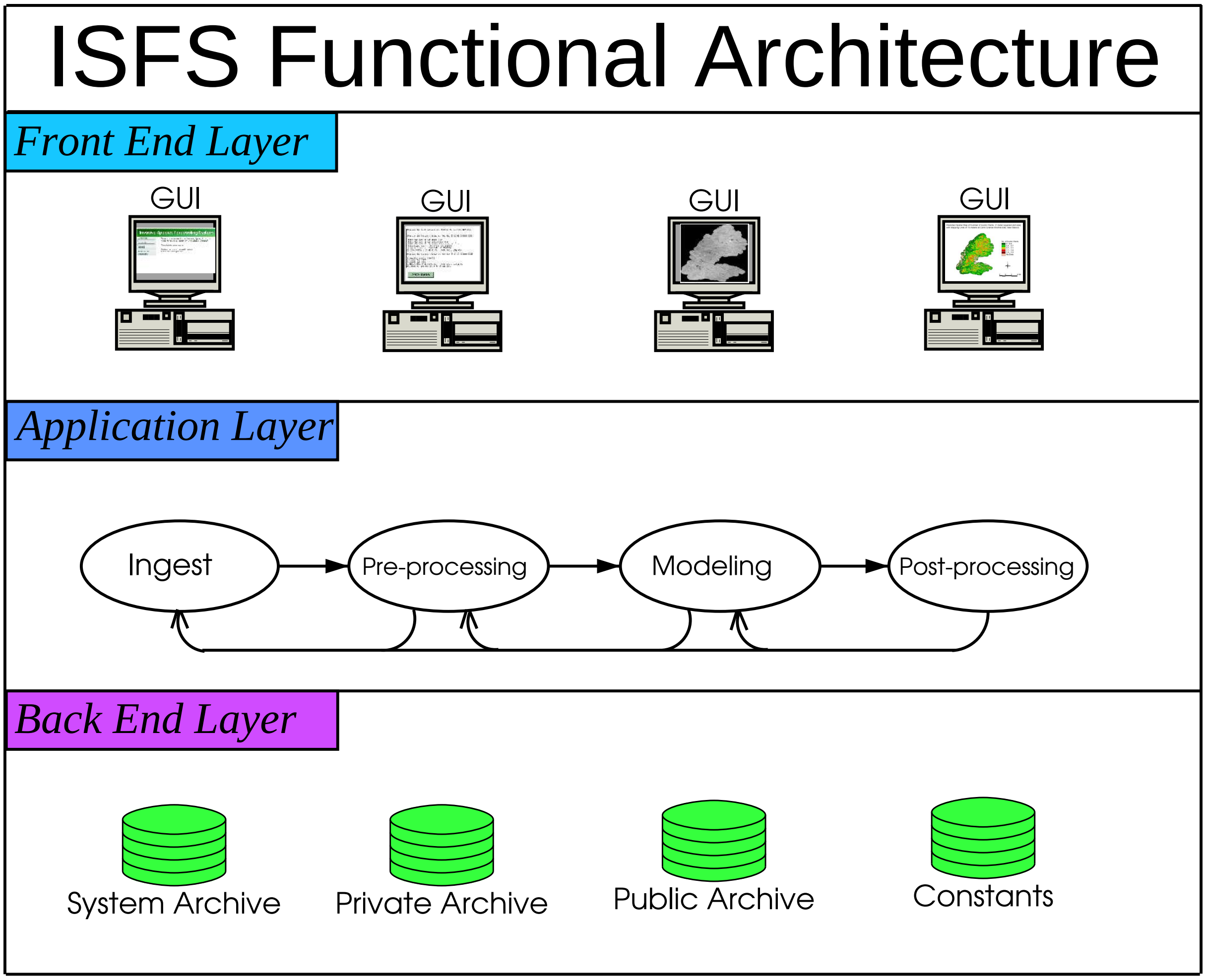


Figure 2. ISFS Functional Architecture

3. A Backend Layer that provides persistent archive storage for both system and user needs.

Each of these major layers and their subsystems are explained in greater detail in the sections that follow. Figure 3 depicts the flow through the system.

The ISFS front end will support a variety of interfaces that allow controlled and tailored access to the various subsystems of the overall system. The front end consists of a graphical user interface subsystem with both client- and server-side components.

2.3.3 Front End Layer

2.3.3.1 User Interface Subsystem

The User Interface Subsystem provides a way of managing information and performing analyses by means of dynamically constructed user workspaces, or role-based views.

A profile database maintains a profile of each user's status and preferences. Activities are implemented by a library of routines accessed through the controls on a web based forms interface. Each user

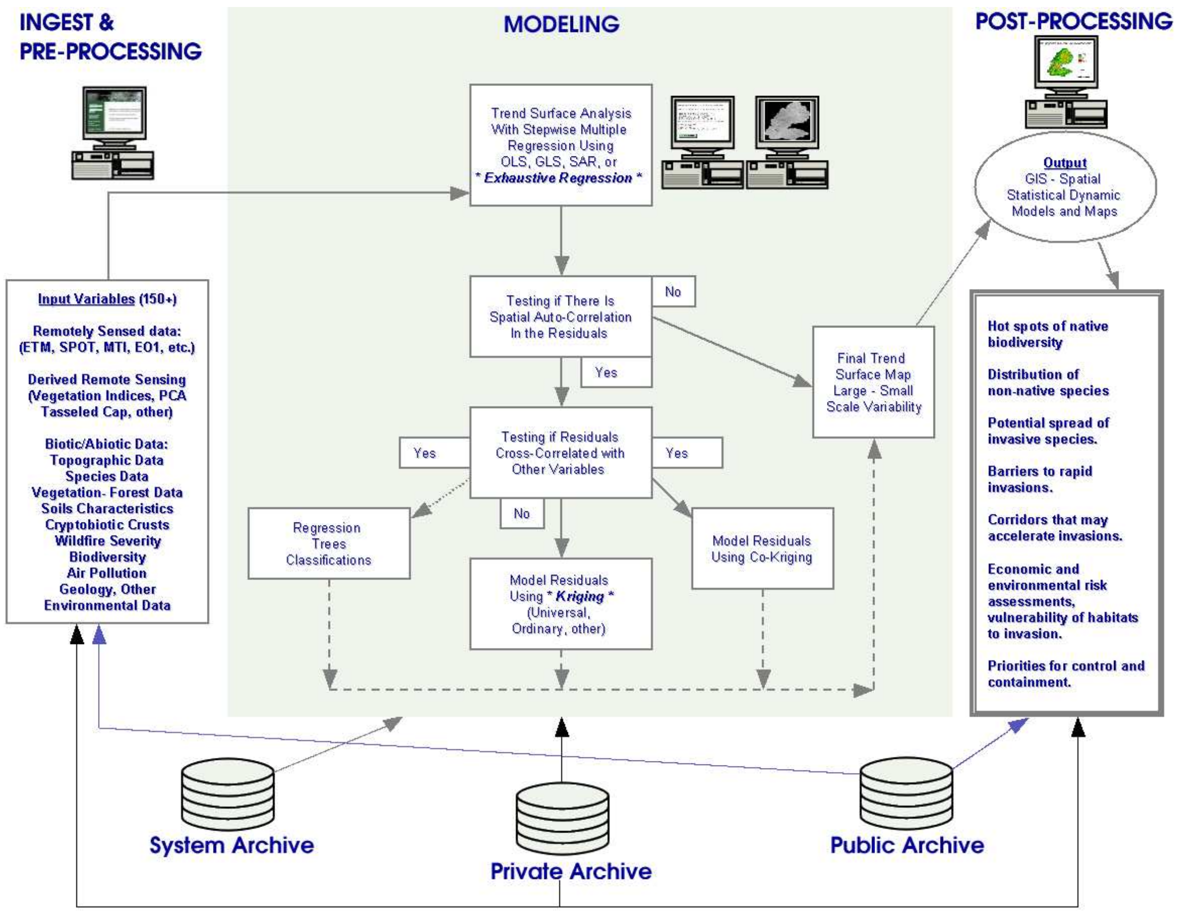


Figure 3. System Flow Chart

is assigned a particular role, which enables them to perform certain actions but not others, and to view certain types of information but not others. Roles include the following:

- “Administrator” — This role provides complete access to the ISFS web system. An administrator would be able to manage other users and have access to their workspace, preferences and authorization information.

- “Model Builder” — This role is intended for the user that wishes to configure and tailor the model related tasks. The Model Builder is someone who has been authenticated and authorized to upload and ingest data in preparation for other users to perform repeated model runs. The Model Builder is a user who has registered with the system and maintains an active logon name and password to access the system. The Model Builder maintains a profile within the system that remembers the Builder’s data selections and makes those selections available to the Builder upon login.

- “Model User” — This role is intended for a user to select specific input data and run it through a previously configured ISFS model run. The Model User is a user who has registered with the system and maintains an active logon name and password to access the system. The Model User maintains a profile within the system that remembers the User’s data selections and makes those selections available to the User upon login.

The roles will be maintained by expanding on the features of PostgreSQL database capabilities, or can incorporate the Lightweight Directory Access Protocol (LDAP). LDAP is a protocol for accessing online directory services over the TCP/IP network protocol, and can be used to access standalone LDAP directory services or directory services supporting the X.500 standard. It provides a standard way for Internet clients, applications and Web servers to access directory listings of thousands of Internet users (http://wp.netscape.com/newsref/pr/newsrelease126.html).

2.3.4 Application Layer

The application layer consists of four subsystems that support the activities, computations, and workflows of the ISFS: Ingest, Preprocessing, Modeling, and Postprocessing. These subsystems may be invoked by the user sequentially, arbitrarily, and iteratively to support the various steps that make up the ISFS modeling process. They may also be invoked automatically by using processing scripts as explained below.

2.3.4.1 Ingest Subsystem

The ingest subsystem will serve as the initial "entry point" for all data used in the system. The data fall into the three main categories: field point measurements, imagery, and ancillary layers. These categories will be further defined in the Requirements Analysis Document. Common to the three categories is that all data ingested into the system will be associated with some geographic location. There will be a validation step to verify the integrity of the data before ingest. Software will be engineered to test for integrity and data authenticity.

Within the subsystem, users will be able to upload field data in a tabular form using standard templates, or tools provided by the Invasive Species Forecasting System. All system required fields will be captured and will be in an accessible database format. Satellite data will be included primarily from external satellite data archives but also user-supplied satellite data or airborne imagery may be used. The primary source for ancillary layers will initially be USGS, however the ingest system will allow user-supplied ancillary layers to be incorporated into the system. Data ingested into the system fall into three categories:

1. A Tabular file (typically from field observations and containing latitude, longitude, date information, and observed values at the given point)

2. Raster layers (typically image data or GIS "grids" which extend over a large area and have one or more layers of information and can have different spatial resolution for the pixel size).

3. A boundary area specifying the region of interest (such as boundary of a national park or monument). The area can be defined by a vector GIS file, a binary raster image, or simply a list of bounding coordinates. The system will allow only one tabular file and one boundary area but multiple raster data files for each modeling scenario. While the upper right, upper left, lower right and lower left may be too close to distinguish with the given GPS accuracy (that is, they would all be reported as the same values), this format would allow larger plots to explicitly define the given area (as opposed to a single point) and thus allow more explicit extraction of the image data for an appropriate area. The format for the tabular data will be a comma delimited ASCII file containing the following elements for each ground observation (hard returns are given here for clarity in the document, the actual values would be on one line):

    - siteID,

- upper left X, upper left Y,
- upper right X, upper right Y,
- lower left x, lower left y,
- lower right x, lower right y,
- large plot ID (if the site is part of a larger plot, if not fill with something like -999)
- date of collection,
- observed variable 1, ..., observed variable N

Mechanisms for data acquisitions will be secured ftp-push for any user supplied data or automated secured ftp pull from the external archives. Users will be issued usernames and will be authenticated through passwords. User activity and preferences will be logged and archived in the system.

A specific list of external archives and required data sets will be established and maintained as part of the ingest subsystem. The interfaces to these archives will be negotiated and a thorough understanding of source and target schema will be included in the interface agreement.

For establishing the baseline canonical example, we assumed that the datasets exist and are stored locally on our servers. Once the HPC technology has been applied and proven we can expand our ingest routines to include additional themes (ancillary layers) and interfaces with government and university databases. Formal procedures for ingest of these data will be documented as the data/source specific requirements become evident. As a general standard operating procedure, we will capture metadata and QA type data that will describe each file to be ingested and write that into the file header or store it in a database record associated with the unique file identifier.

2.3.4.2 Preprocessing Subsystem

The pre-processing stage manipulates the data resulting from the ingest stage into a format/structure that can be used by the modeling component of the system. Limited pre-processing is needed for the tabular data, since the format will be specified in the ingest phase.

The subsystem may perform resampling if the input raster layers are not at the same resolution. The merged data product will be written to the archive in a common analysis format, possibly GeoTIFF.

The primary component of the pre-processing stage will be to extract, for each site (as defined by the coordinates in the tabular file), the information from each raster layer for that site. These values extracted from the raster layers will be appended as columns to the ingested tabular file. The results of this pre-processing step will be referred to as the *"merged tabular data"*. A model builder selects items from the merged data set to build the model array.

A secondary component of the pre-processing stage will be to create new variables from existing columns of the merged tabular file. These new variables will either be pre-programmed or user-defined functions of the existing columns. The results of this step will be referred to as the *"merged, augmented tabular file"*. Either a "merged tabular file" or a "merged, augmented tabular file" is required to go on to the modeling stage.

2.3.4.3 Modeling

Modeling in the first version of the ISFS will be empirical in nature and utilize statistical techniques. The general theme of the modeling will be to predict a certain species migration through or invasion of habitat based on remote sensing imagery and ancillary data layers. The modeling subsystem will support five basic workflow activities:

- *Step 1 — Data Array Construction.* Each model will require an array containing the response, or dependent, variable (the "Y" variable) and a set of predictor, or independent, variables (the "X" variables). Constructing the data array needed for modeling will start with a set of geographic coordinates. These will likely come from the geographic coordinates associate with the tabular field data of interest. In addition to the coordinates, the Y variable will be extracted from the tabular field data, either directly or as a function of one or more elements in the field data. The X variables will come from any of the three data sources. X variables from the field data can also be directly extracted or be a function of one or more elements from the data. X variables will be extracted from the satellite and ancillary data by using the coordinate information from the field data to extract values from the imagery for the corresponding pixels. These X variables can come directly from the satellite or ancillary data or be a function of one or more satellite or ancillary variables. A diagram of the modeling array is shown in Figure 4.

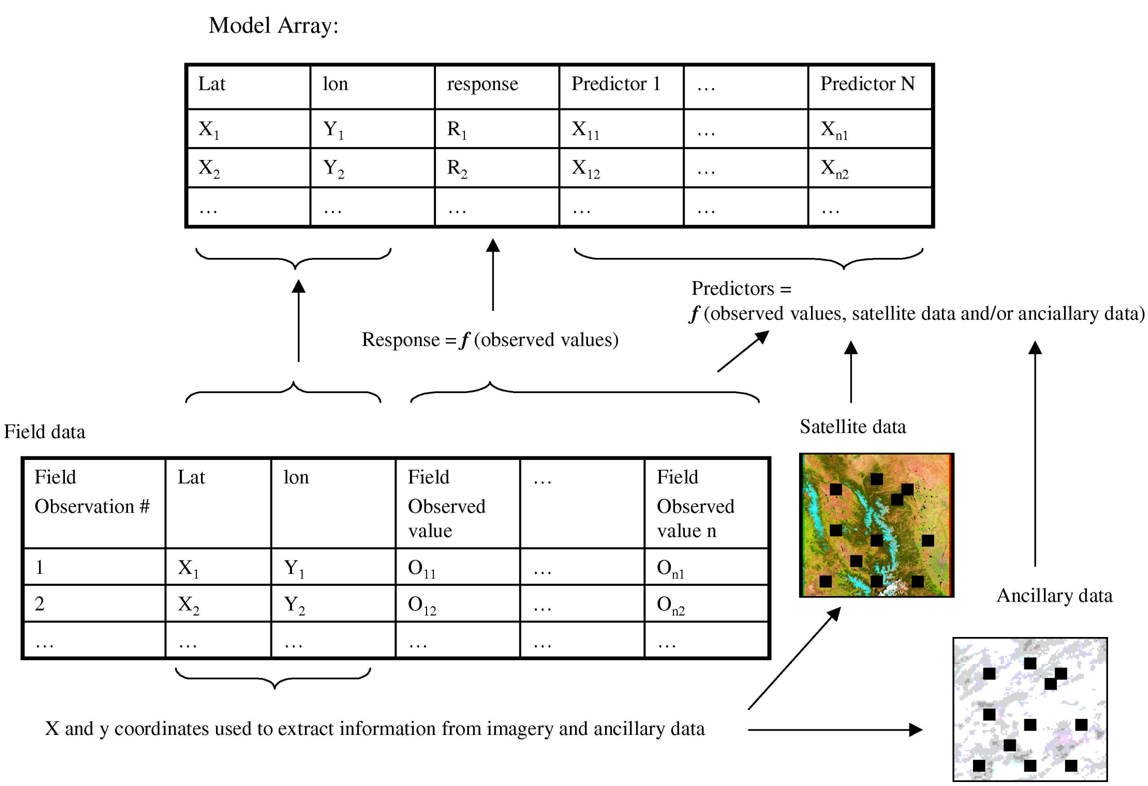


Figure 4. Schematic of Modeling Array

- *Step 2 — Model Selection and Fitting.* Once the array is constructed, model selection will involve screening X variables to see which are most related to the Y variable. Methods to be considered are graphical exploratory analysis, stepwise regression, and combinatorial screening. The variables found to relate to the Y variable will be related through statistical models. Models to be considered fall into the generalized linear models framework using generalized least squares and regression tree models and others as applicable.

- *Step 3 — Model Diagnostics.* Model diagnostics will include assessing the fit of the model and testing the assumptions implicit to the model. Of particular interest will be the spatial nature of the data and assumption of spatial independent or, alternatively, accounting for spatial dependence within the models.

- *Step 4 — Model Acceptance/Adjustment/Refinement.* Results from the diagnostics will be used to either confirm the appropriateness of the model or influence adjustments or refinements to the model. Adjustments or refinements will require returning to step 2, model selection and fitting.

- *Step 5 — Model Output.* Once an appropriate model is accepted, model output will include the model formula itself as well as metadata describing the user responsible for the model selection and the data used to drive the model.

2.3.4.4 Post-processing

The post-processing subsystem uses outputs from the Modeling Subsystem to generate visual and graphical products that are then made available to the user. In a typical scenario, the results of the OLS regression are applied to the input satellite and DEM data to provide a preliminary map of the total plants in the region. Kriged estimates of the residuals of this regression are then added to this map to produce an improved map. The postprocessing subsystem will typically produce a standard, application interchangeable output file, such as an IDL file with ENVI header information, that can be used by other applications to reproject the data and overlay with other data layers as requested by the user. The final data products will be packaged with the appropriate metadata, assigned a unique data set identifier, and archived.

2.3.5 Backend Layer

The backend layer provides persistent storage for both system and user needs.

2.3.5.1 Archive Subsystem

The ISFS backend consists of an archive subsystem logically comprising system, private, and public partitions. Subsystem control will be coordinated by a database that will store pointers to archived files. Initially, the files that we maintain will be contained in a logically arranged directory structure and indexed with a unique file ID. For externally stored data, the archive system will store a file ID and pointer or URL that can be used to retrieve and stage the archived files for subsequent processing.

2.4 Design Status

The design status section includes the following categories:

1. Constraints and Concerns
2. Assumptions
3. TBD Requirements
4. Model Code Performance Improvement

2.4.1 Constraints and Concerns

The development environment uses ssh to access the CT cluster at Goddard. Firewall constraints will require a TBD alternative approach for the next round of development and operational deployment.

2.4.2 Assumptions

This section will be completed as assumptions arise, including:

- SSH will not be part of the development environment going forward
- The ingest subsystem will monitor the number and volume of data brought into the system, with an ability to break this down by location, user, and external archive. Formal procedures for ingest of these data will be documented as the data/source specific requirements become evident. As a general standard operating procedure, we will capture metadata and QA type data that will describe each file to be ingested.

2.4.3 TBD Requirements

The following key points are to be analyzed at the beginning of the next phase of development. Each of these items is essential to the success of the project and will be documented for design of the system during the new development cycle. These items will be identified in the next SDD:

- The kriging algorithm that is run on the cluster will need to be well defined.
- The IDL code will need to be maintained.
- The application will have to be developed to be multi-user safe.
- Pre-processing needs to be well defined and developed.
- The new project cluster to be built Fall 2003 will allow the Java Remote Method Invocation (RMI) to replace the current JCraft ssh scheme, allowing for a well-defined interface. The current transport is not strong enough to handle the data load the next phase of the project requires.
- Exception handling in the Java code will need to be further defined.
- There needs to be a data clean-up scheme that specifies what data is to be discarded, what data is to be kept, and for how long.
- Internal storage requirements, including description of arrays, their size, their data capacity in all processing modes, and implied limitations of processing need to be determined.
- Development cycle milestones will be developed according to software engineering principles including analysis, design and requirements. Checkpoints between all phases of development will be scheduled to take place throughout the development cycle to assure all modules of the application are working well with each other as the development timeline moves forth. Checkpoints for the front-end, middleware, back-end and data modules of the project are necessary to avoid any potential delays in bringing all parts together as the end of the development cycle nears. The milestones and checkpoints will be rolled into the project schedule.
- Further design details for each module of the next development cycle will continue to be analysed and included in this document prior to development.
- There needs to be a job controller for managing model runs on the cluster. Cluster management tools to work with the job controller and ISFS need to be well defined.

2.4.4 Model Code Performance Improvement

We are working with three "canonical" study sites: the Cerro Grande Fire Site in Los Alamos, NM (CGFS), Rocky Mountain National Park, CO (RMNP), and Grand Staircase Escalante National Monument, UT (GSENM). The three sites provide contrasting ecological settings and analysis challenges and vary in the types and scales of data used, areas covered, and maturity of the investigation. As described in detail in the Baseline Software Design Document (BP-BSD-1.3), three factors influence the performance of ISFS model code: the size of the output surface area over which kriging occurs (area), the total number of number of sample points in the data set (pts), and the number of "nearest neighbor" (nn) sample points from the total data set actually used to compute a kriged value for any given point in the output area. When we first began work with colleagues at USGS, a scalar, single-processor run of this model using S-plus took approximately two weeks. The major computational bottleneck in the model is the kriging routine. Solving for the weights in the equations that form the ordinary kriging system uses LU decomposition with backsubstitution to do matrix inversions. The overall computational complexity of ordinary kriging is thus $O(n^3)$, and the time required to compute a result is strongly influenced by the number of sampled data points used to estimate the residual surface across the entire study area.

The overall goal for code improvement is to reduce processing times and increase the amount of data handled by the model. As described in BP-BSD-1.3, increasing the amount of data handled by the model translates into either increasing spatiotemporal resolution or increasing coverage. We first wish to accomplish quantitative improvements in the underlying model that have been agreed upon by the user community as minimal advances needed to improve core capabilities. These goals were driven by the fact that we are building a 32-node cluster in the USGS facility. We refer to these as "Community Improvement Goals." The ESTO/CT program, however, provides access to even greater computational capabilities that can be used to apply this modeling approach to some important and challenging problems that have heretofore been unapproachable. We would therefore like to use CT's clusters to attain more challenging performance improvement goals at the same time we are accommodating basic needs. We refer to these complimentary challenges as "Advanced Improvement Goals." Table 1 provides a summary of the baseline performance characteristics of the model code as well as the various performance goals anticipated over the course of the project.

2.4.4.1 Milestone F — First Code Improvement (Parallel Kriging)

Kriging is a spatial interpolator that determines the best linear unbiased estimate of the value at any given pixel in an output surface or image using a weighted sum of the values measured at arbitrary sample locations. It determines the weights and the spatial continuity of the data as measured by the variogram. The scalar kriging algorithm is a double loop over all rows and for each pixel within the row. At each pixel we determine the n nearest neighbor sample points and compute the $(n \times n)$ distance matrix containing the Euclidean distance between each sample points, and also compute the $(n \times 1)$ distance vector from the pixel to each of the sample points. The Euclidean distances are converted to statistical distances by applying the variogram model to create a covariance matrix and vector. We obtain the kriging weights by multiplying the inverse of the covariance matrix by the covariance vector. The computationally expensive part of kriging is the inversion of the covariance matrix, which is done at each pixel since the nearest neighbor sample points can vary across the kriged surface.

The steps to estimate the value at each pixel are independent of all other pixels. The algorithm is therefore 'elegantly parallel' and highly amenable to parallel implementation via domain decomposition; we simply assign to each processor a section of the output kriged surface or image. We chose to decompose the domain along the rows only, i.e. each processor works with full rows of the output surface. This means we can leave unaltered the inner loop over columns. We could decompose into contiguous rows,

Table 1. Current performance characteristics and improvement goals.

BASELINE SCENARIO		**Sec**	**Min**	**Hrs**	**Days**	
CGFS_base_079_pts_79_nn_01x_area (S-Plus) (Version 0.0)		-	-	-	-	
CGFS_base_079_pts_18_nn_01x_area (S-Plus) (Version 0.0) (USGS Actual)		1209600.0	20160.0	336.0	14.0	
CGFS_base_079_pts_18_nn_01x_area (S-Plus) (Version 0.0) (NASA Estimate)		1608426.0	26807.1	446.8	18.6	
CGFS_base_079_pts_18_nn_01x_area (FORTRAN) (Version 0.1)		114.5	1.9	0.0	0.0	
CGFS_base_079_pts_79_nn_01x_area (FORTRAN) (Version 0.1)		4702.6	78.4	1.3	0.1	A
RMNP_base_1180_pts_18_nn_01x_area (FORTRAN) (Version 0.1)		443.0	7.4	0.1	0.0	
RMNP_base_1180_pts_1180_nn_01x_area (FORTRAN) (Version 0.1) (est.)		6812384.0	113539.7	1892.3	78.8	B
COMMUNITY IMPROVEMENT (CI) GOALS	**x baseline**	**Sec**	**Min**	**Hrs**	**Days**	
CGFS_base_079_pts_79_nn_01x_area (Version 1.0 - F)	25.0	188.1	3.1	0.1	0.0	C
CGFS_base_790_pts_79_nn_01x_area (Version 2.0 - G)	25.0	188.1	3.1	0.1	0.0	D
CGFS_base_790_pts_79_nn_10x_area (Version 2.0 - G)	2.5	1881.0	31.4	0.5	0.0	E
ADVANCED IMPROVEMENT (AI) GOALS	**x baseline**	**Sec**	**Min**	**Hrs**	**Days**	
CGFS_base_079_pts_79_nn_01x_area (Version 1.0 - F)	200	23.5	0.4	0.0	0.0	F
RMNP_base_1180_pts_1180_nn_01x_area (Version 2.0 - G)	1000.0	6812.4	113.5	1.9	0.1	G
RMNP_base_11800_pts_1180_nn_01x_area (Version 2.0 - G)	1000.0	6812.4	113.5	1.9	0.1	H
RMNP_base_11800_pts_1180_nn_100x_area (Version 2.0 - G)	10.0	681238.4	11354.0	189.2	7.9	I

A Proposed CGFS canonical baseline using FORTRAN kriging routine.
B Proposed RMNP canonical baseline using FORTRAN kriging routine.

C Milestone F CI Goal - speed up - 75% efficiency, 32 node cluster = 25x speed up
D Milestone G CI Goal - increased resolution - “sliding window” adaptive selection of 10% of 10x nn from 1x area
E Milestone G CI Goal - increased coverage - “sliding window” adaptive selection of 10% of 10x nn from 10x area

F Milestone F AI Goal - speed up - 75% efficiency, 256+ node cluster = 200x speed up
G Milestone G AI Goal - speed up - 75% efficiency, 1024+ node cluster = 1000x speed up
H Milestone G AI Goal - increased resolution - “sliding window” adaptive selection of 10% of 100x nn from 1x area
I Milestone G AI Goal - increased coverage - “sliding window” adaptive selection of 10% of 100x nn from 10x area

effectively giving each processor a strip of the output image. Instead, we chose to assign consecutive rows to separate processors. Thus, for a kriging 512×512 image using 32 processors, the first processor would be assigned rows 1, 33, 65, …, 449 and 481, while the last processor would calculate rows 32, 64, 96, …, 480 and 512.

Both domain decompositions are equally load balanced if the number of sample points used in the covariance matrix is always the same at each pixel. This is the case now, but soon we plan to implement an adaptive scheme that will use more points in densely sampled regions and fewer points in sparsely sampled areas. Significant load imbalance would result if we assigned sparsely sampled rows to one processor while assigning densely sampled rows to another processor.

We have implemented parallel kriging in FORTRAN using MPI, the Message Passing Interface. Our code employs a ‘node 0’ controller process and a collection of worker nodes. Prior to execution we copy to each node an input data file containing the dimensions and cell spacing of the output kriged surface, the variogram parameters that describe the spatial structure, and the series of plant diversity measurements (UTM X and Y coordinates and the number of plant species at each location). Each node reads this input data file, computes the kriged estimates for its assigned rows, and then sends each row to node 0. Node 0 only receives the data from the worker nodes, assembles the kriged surface in memory, and writes the final kriged estimates to its local disk.

We overlap the computation with the communication to increase parallel efficiency. When the first row has been calculated, we issue an asynchronous send (MPI_ISEND) of this row to node 0. Since this is a non-blocking send, the processor proceeds to calculate the second row. At the end of this row, we issue a wait (MPI_WAIT) to insure that the first row has been received by node 0 before proceeding. For the smallest kriged surface we tested (512×512) the compute time for each row is over 4 seconds — thus the first row has more than sufficient time to be received and the wait call should also return 'immediately' (in reality, the latency time MPI's implementation of the MPI_ISEND and MPI_WAIT calls). Meanwhile, node 0 posts a serial set of asynchronous receive calls (MPI_IRECV) for each row sent by the worker nodes, followed by a series of waits (MPI_WAITS). When the waits are finished, each row of data is copied into the appropriate location within the output kriged array on node 0.

We have tested our parallel implementation on the Thunderhead cluster at NASA Goddard Space Flight Center. Thunderhead is a 512-processor, 2.4 GHz Pentium 4 Xeon cluster with 256 GB of memory and 20 TB of disk storage. Each node is connected to the others with dual-port Myrinet. Node 0 is a Linux PC with a single 1.2 GHz AMD Athlon processor and 1.5GB memory, which resides on one of our desks and is connected to the Medusa cluster via fiber Gigabit Ethernet. We typically only log into Node 0, and to the user it appears that all calculations are done on Node 0.

2.4.4.2 Parallel Kriging Results

We ran four test problems to evaluate the efficiency of the parallel implementation. We held the number of input data points constant at 79 (the size of the field sample data set for Cerro Grande), while the output kriged image size varied from 5122, 7682, 10242, to 20482. Table 2 shows the results of our preliminary timing study. The processing times shown are elapsed wall-clock time in seconds. As expected, the kriging time increases in direct proportion to the area of the output kriged surface (e.g. the 20482 problem ran 16x longer than the 5122 case). The processing times decreased nearly linearly as the number of processors was increased, as shown in Figure 5. We define the scaling efficiency for N processors as the ratio of the 1-processor to N-processor wall-clock times divided by N. The efficiencies we obtained were excellent, shown in Table 3, ranging from 96–98% when using 32 processors and over 99% when using 16 or fewer processors. The scaling efficiencies dropped slightly for the 64 processor tests, but were still greater than 97% for the 20482 problem.

Overall, we improved the run time for the Cerro Grande test to 2 minutes and 13 seconds, which is significantly better than our Milestone F Community Improvement goal of 3 minutes and 8.1 seconds. We improved the Rocky Mountain test to 19.5 seconds, which allows for nearly interactive response. Figure 6 shows an example output map produced by the modeling process.

Table 2. Timing Results (Elapsed Wall Clock Seconds)

Number of Processors	Size of Kriged Image			
	2048^2	1024^2	768^2	512^2
64	583.9	147.5	84.0	38.4
32	1150.4	289.8	163.8	73.8
16	2285.1	573.89	324.0	144.7
8	4558.4	1142.0	642.8	287.2
4	9083.9	2277.4	1281.3	571.5
2	18190.4	4556.3	2562.0	1140.9
1	36252.7	9079.6	5107.0	2269.1

Table 3. Scaling Efficiencies

Number of Processors	Size of Kriged Image			
	2048^2	1024^2	768^2	512^2
64	97.0%	96.2%	95.0%	92.3%
32	98.5%	97.9%	97.4%	96.0%
16	99.2%	98.9%	98.5%	98.0%
8	99.4%	99.4%	99.3%	98.8%
4	99.8%	99.7%	99.6%	99.3%
2	99.6%	99.6%	99.7%	99.4%
1	100.0%	100.0%	100.0%	100.0%

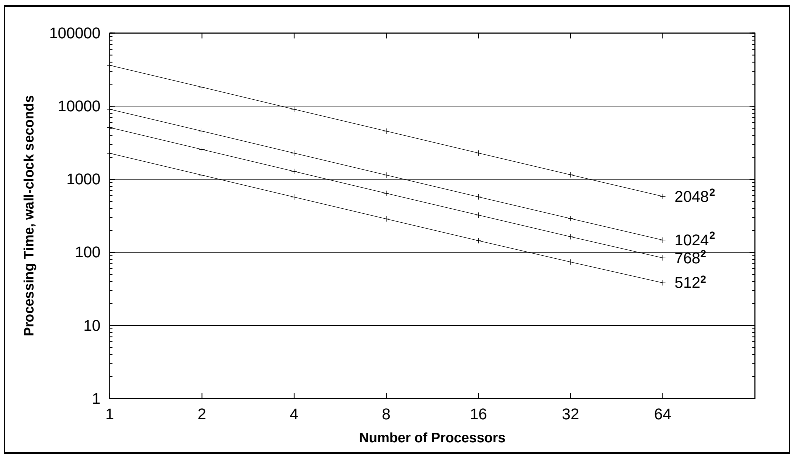


Figure 5. Scaling Curves on Medusa

The canonical output size for Cerro Grande is 715×652. To understand the scaling behavior of our code when applied to larger problems, we examined analytical problem sizes varying from 5122 to 20482. Figure 7 shows that we can scale nearly linearly to 256 processors, thus meeting this aspect of the Milestone F Advanced Improvement goal. The curve in Figure 7 shows results for the 1024×1024 case, which is one of the analytical cases run on frio/medusa. We can run the 715×652 case on 256 nodes of thunderhead in 49.5 seconds. This is within 2x of our goal of Milestone F Advance Improvement goal of 23.5 seconds. However, we see that we are running 2.3x slower on thunderhead than on frio/medusa, indicating that when adjusted by a factor of 2.3, we beat our Advanced Improvement goal substantially. We anticipate even better performance on the processors we are building into our clusters for delivery to USGS.

2.4.4.3 Summary

These results indicate that we have achieved our Milestone F Community Improvement goal of a 25x speed up on a 32-processor cluster with greater than 75% efficiency. The results also document the general scaling behavior of the kriging algorithm and point to the limits in scalability one might expect as more nodes are allocated to the canonical data sets.

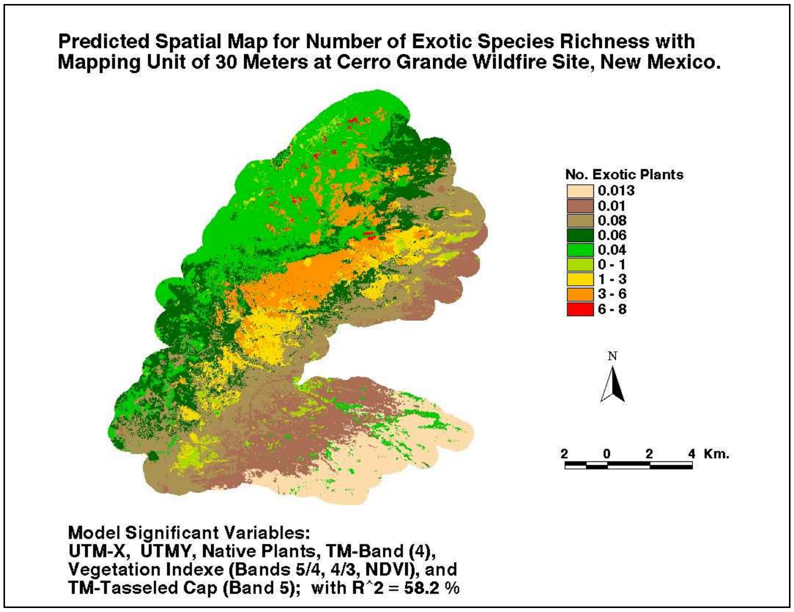


Figure 6. Predicted exotic species richness on the Cerro Grande Wildfire Site, Los Alamos, NM. This is an example of the type of predictive spatial map used by USGS in invasive species decision support.

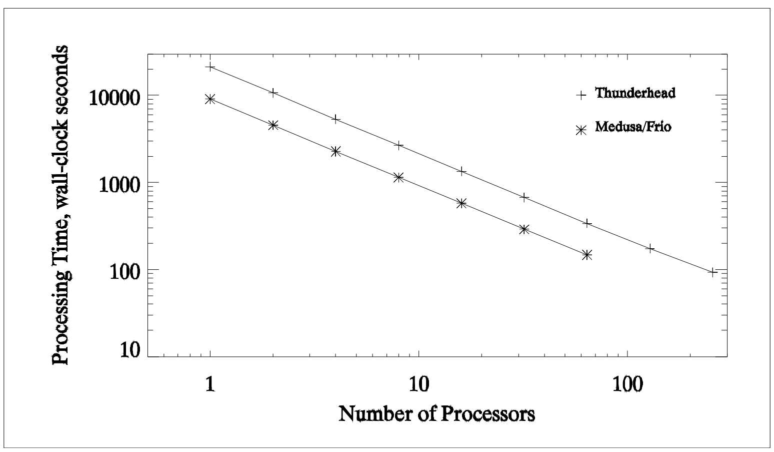


Figure 7. Comparison of Scaling Curves on Medusa and Thunderhead

2.5 Development Environment

ISFS is being developed primarily using Java and other open source software on a Linux platform. Table 4 depicts various software tools as they relate to their particular area of the development environment.

Software included in the ISFS development environment includes the following:

- Ant — ISFS uses Ant as a build tool. ANT can be downloaded from http://ant.apache.org/
- Beowulf Cluster — Parallel computer with a cluster of PCs running connected by its own private LAN.

Client	Web Engine	Database & Cluster
Web browser	Tomcat Jakarta Struts Poolman J craft JOX	PostgreSQL ENVI IDL Beowulf Cluster

Table 4. Software Development Environment

- Bugzilla — The Bugzilla Defect Tracking System tracks bugs and enhancement requests.
- CVS — Concurrent Version System configuration management software is used for file backup and version control of developers' files including all .html pages, .jsp's, scripts, images, and documentation.
- ENVI — Remote sensing software that includes hyperspectral analysis.
- IDL — (Interactive Data Language) software is used for data analysis, visualization, and cross-platform application development. IDL is the underlying language for ENVI.
- Jakarta Struts — ISFS is using the Jakarta Struts web application framework for design and implementation. Struts handles the J2EE code, JSPs, servlets, and Java code. Details about Struts can be found at http://jakarta.apache.org/struts/.
- Java — J2SE 1.4 is required and is available at the Java website: http://java.sun.com/
- JCraft — SSH2 Terminal Emulator in Pure Java — http://www.jcraft.com/jcterm/
- JOX — "JOX is a set of Java libraries that make it easy to transfer data between XML documents and Java beans." — http://www.wutka.com/jox.html
- Poolman — The PoolMan library and JDBC2.0 Driver and DataSource provide a JMX-based, XML-configurable means of pooling and caching Java objects, as well as extensions for caching SQL queries and results across multiple databases.
- PostgreSQL — The database used for ISFS is a Postgres database and the name of the database is isfs. The schema file is located in the /BP/bin folder and is named isfs.sql. This is an SQL script used to initialize the database.
- TogetherJ — TogtherJ is the project's UML/IDE tool. The TogetherJ project file, ISFS.tpr, is located in the BP folder of the project.
- Tomcat — Tomcat provides the Servlet and JSP engine that ISFS uses to run the Portal. To get Tomcat, the open source software can be downloaded from http://jakarta.apache.org/tomcat/. Currently the project is using Tomcat 4.1.18, but a more current version should work fine. The web.xml file for ISFS is located in the /BP/config folder in case you need to alter that file.

3 Operations Overview

3.1 Operations Scenarios

The following diagram illustrates the current scenario of executing a model run using ISFS. For a detailed discussion of the modeling approach please refer to the Baseline Software Design Document, BP-BSD-1.3.

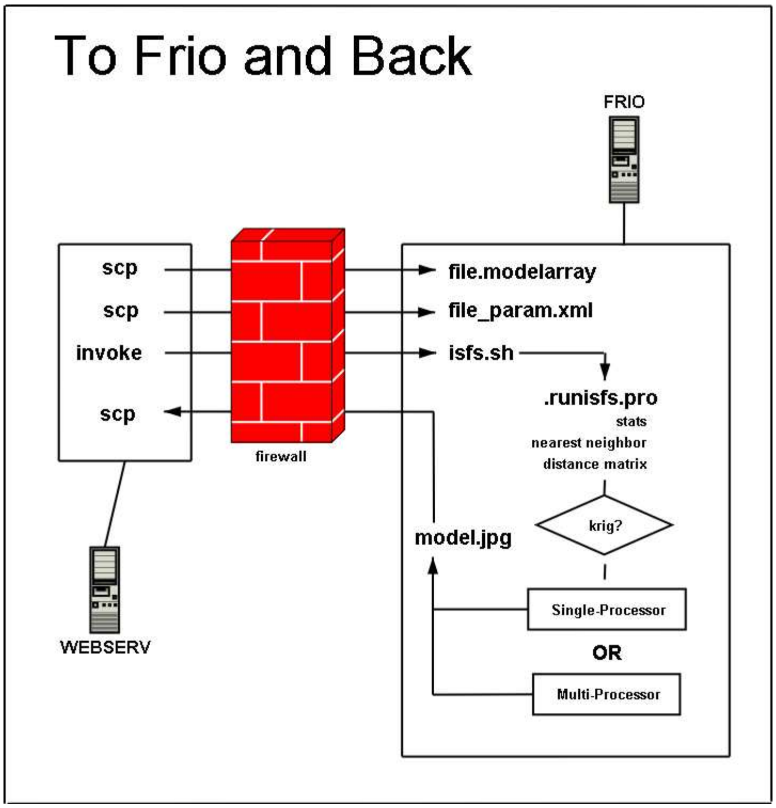


Figure 8. To Frio and Back

The current environment has been developed to handle communication through the firewall between the development server "webserv" and the project's data server "frio" which is our interface with the Goddard cluster. The scenario involves:

1. A secure copy of data files to frio through the firewall.
2. The script —texttisfs.sh is invoked on frio to compute stats, nearest neighbor, and distance matrix.
3. On frio, the processing and kriging is completed using the Beowulf cluster
4. The resultant dataset and images are then secure copied back to webserv for display to the user.

3.2 System Performance

The performance of the ISFS project is tied into the cluster. The cluster is a **Beowulf**-class parallel computer, (a **cluster** of PCs running LINUX, connected by its own private LAN.) Two dedicated clusters for the project are to be built in Fall, 2003 for facilities at GSFC and Colorado State University. See section 2.4.4 for benchmarks to system performance.

Goddard clusters used by ISFS are reviewed in table 5.

Hewlett-Packard/Compaq AlphaServer SC45	• 1,392 processors: 880 (1.25 GHz), 512 (1 GHz) • 696-gigabyte memory • 8.5-terabyte disk • 3.2 teraFLOPS peak
Thunderhead (HIVE III) — PSSC Labs	• 512-processor 2.4 GHz Pentium 4 Xeon cluster • 256-gigabyte DDR random access memory • 20-terabyte disk • 2 gigabit-per-second Myrinet • 2.5 teraFLOPS peak
Medusa (HIVE II) — Custom-Built	• 128-processor 1.2 GHz Athlon MP cluster • 17 coupled developer nodes in GSFC scientist offices • 64-gigabyte DDR random access memory • 2.56-terabyte disk • 2 gigabit-per-second Myrinet internal network • Fast Ethernet network to developer nodes

Table 5. Goddard Clusters

4 Design Description

4.1 Subsystem Capability

The subsystem is currently setup to run canned model runs for the purpose of satisfying project milestones, demonstrate use of the system, and identify requirements for the next phase of development.

4.2 Processing Restrictions

Current restrictions on processing are due to the firewall between the development machines and the cluster.

4.3 Subsystem

The subsystem can be broken into the three areas of Client Browser, Web Applications and the Business Tier.

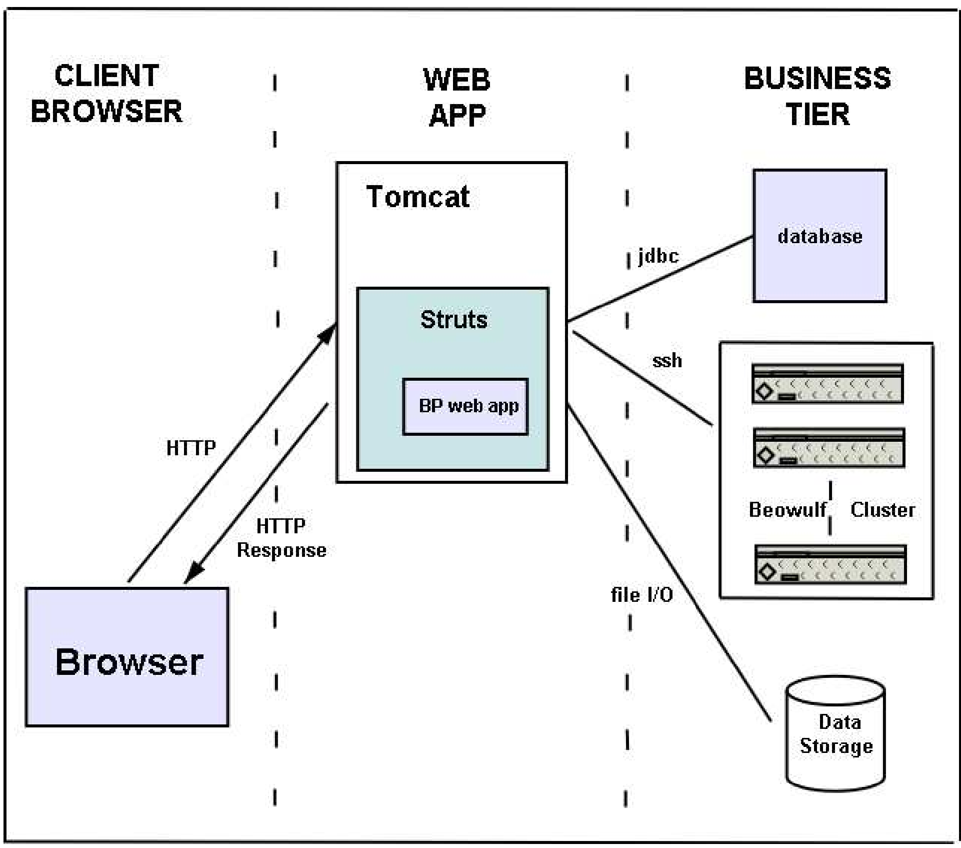


Figure 9. Subsystem

Struts is written in the popular and versatile Java programming language. Java is an object-orientated language, and Struts makes good use of many object-orientated techniques. In addition, Java natively supports the concept of threads, which allows more than one task to be performed at the same time (see "The Java Language and Application Frameworks" http://jakarta.apache.org/struts/userGuide/preface.html#java).

Struts implements a suite of custom tags that are pre-defined within the Struts framework to perform basic functionality that the web applications need such as logon, and processing forms dictated by the business logic.

4.4 Input/Output

The steps in Section 4.5, Use Case 1.0 details the steps for the input and output (I/O) of data for a model run. The following I/O steps are computed on the project's server "frio" from the reading in of data to the output of results:

1. Read in tabular data
2. Compute the distance matrix
3. Map tabular data to remote sensing data
4. Perform stepwise regression
5. Fit squares
6. Engage kriging routine
7. Output results variables
8. Rasterize and output graphical presentation of data as .jpeg (for higher quality images the project is interested in saving these images as GeoTiff).

4.5 Processing

The following use case describes the processing of a request to run an ISFS model:

Use Case 1.0

1. User enters account name and password to enter the system.

2. User is authenticated in lookup table and enters the new system.
   OR
   User is not authenticated and is brought back to the login page with the jsp displaying an error message that reads "*Your Account name or password is invalid*".

3. Action locates list of Study Sites from the DB (or from file system) that are ready to be submitted as a model run.

4. Action locates list of kriging types from DB (or from file system).

5. Using JSP, draw page with list of study sites.

6. User selects study site and submits selection.1. Action locates list of Model Arrays (MA) from DB (or from file system).

7. Using JSP, draw page with Model Array list for the selected study site.

8. User selects model array and submits selection.1. Action locates list of Kriging routines from DB (or from file system).

9. Using JSP, draw page with Kriging routines list for the selected study site.

10. User selects Kriging routine and submits selection.

11. Validate all input:
    (a) Files exists MA (and possibly kriging)
    (b) Limit checks on NN
    (c) Cluster Ready/Available
    (d) Has this setup already been run?
12. Send MA, krig type, NN and a process identifier to FRIO and invoke isfs.sh as '`sh isfs.sh /to../procid/MA krigtype NN procid`'
13. The model runs and produces output in `/from../procid/outputfile.jpg` (for further detail, see Section 4.4)
14. When the model run is completed, a java program executes and places an RMI call back to the controller passing the .jpg, procid and associated metadata (an XML file) back to the controller.
15. The controller is notified and stores the .jpg on the file system. The metadata is entered in the DB and the geotiff is displayed to the browser in an appropriate way. The metadata for the run is displayed (possibly requiring a call to the DB to get it).The user is allowed to add comments to a field within the metadata record. The user views the output image.

4.6 Structure

The following two UML run sequence diagrams detail the launching and running of a model run. Structure charts expanded to the object-oriented level can be found in Appendix B.

4.6.1 Run Model Sequence Level 1

In Figure 10, the sequence diagram illustrates the chronological execution of a user request to run a model after all require input has been entered. Objects are enumerated horizontally by rectangles at the top of the diagram. Edges between objects indicate an access or method invocation between objects.

4.6.2 Run Model Sequence Level 2

Once input parameters have been extracted and validated from user input this sequence is executed to launch the model run and wait for results if the user has chosen to do so. This sequence, shown in Figure 11 is nested within Run Model Sequence Level 1.

4.7 Data Storage

Please refer to Section 2.3.5.

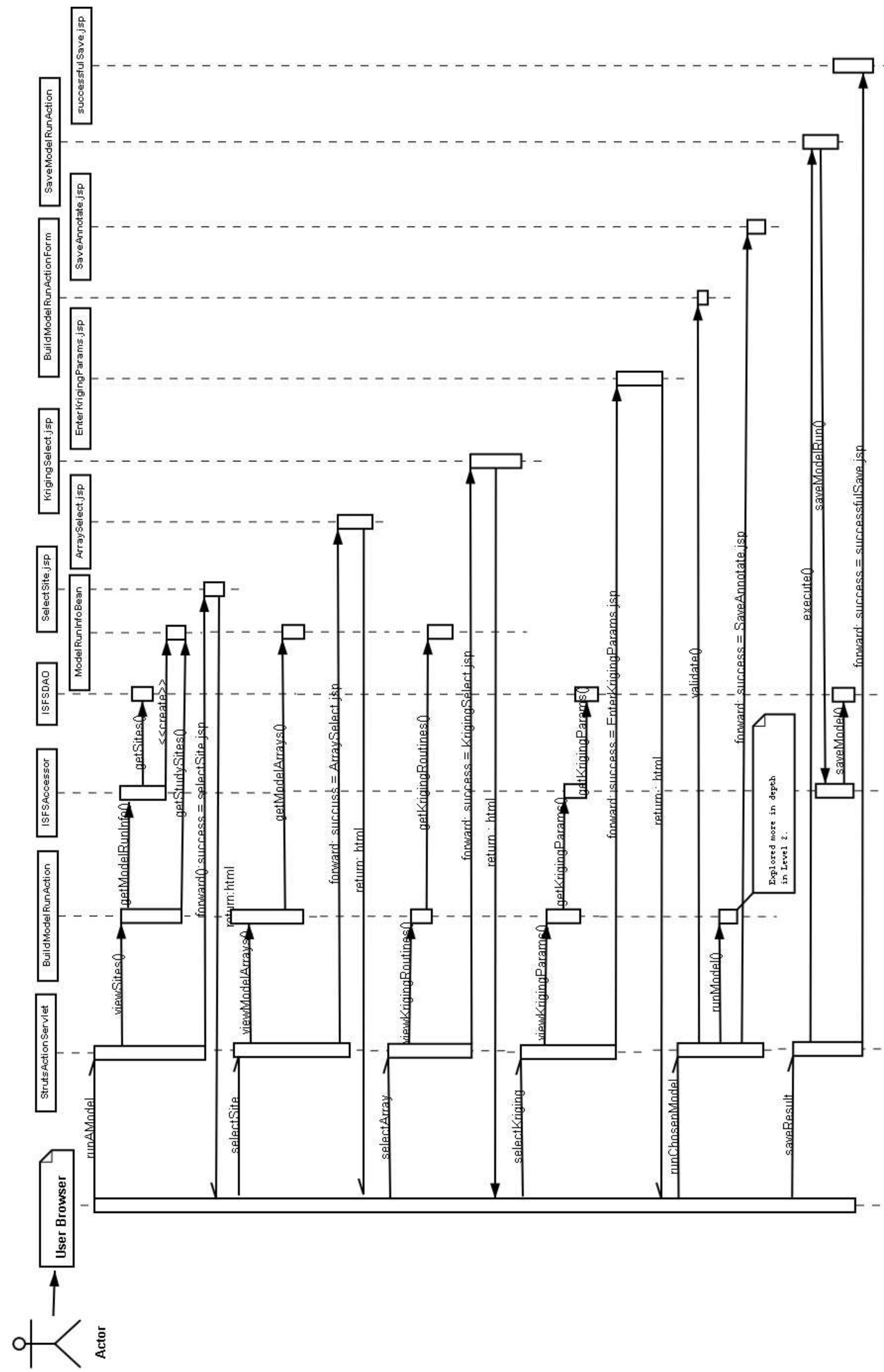


Figure 10. Run Model Sequence Level 1

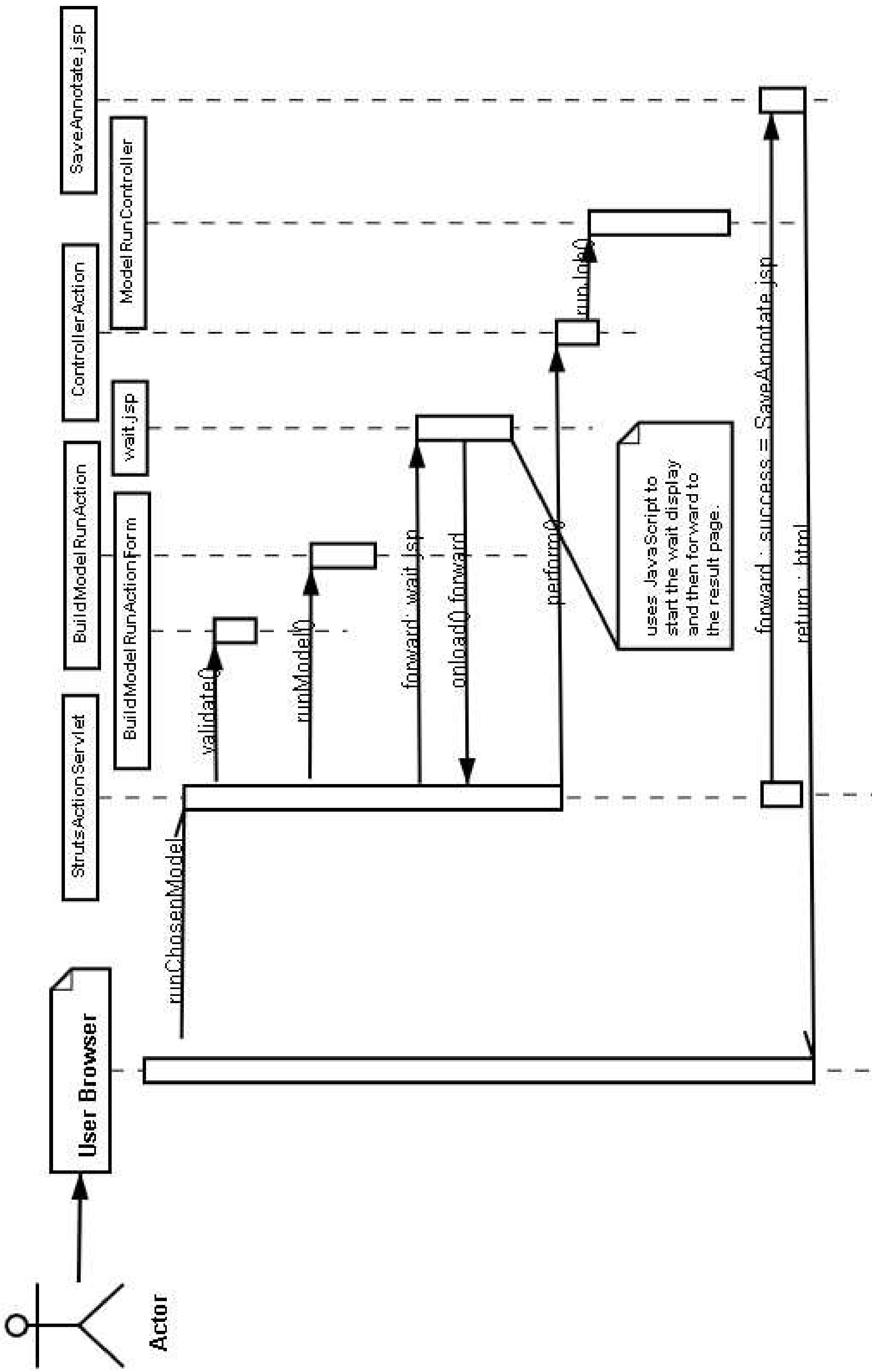


Figure 11. Run Model Sequence Level 2

4.8 Graphical User Interface (GUI)

The GUI for ISFS incorporates selection screens drawn by java server pages via Tomcat. User interface specifications are detailed in Section 2.3.3. The following diagram is based on Use Case 1.0, found in Section 4.5.

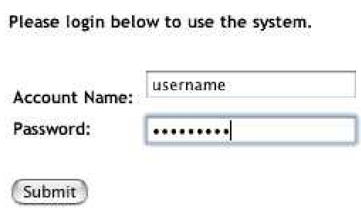
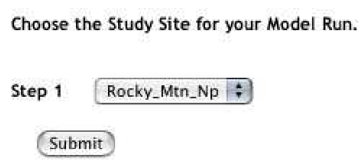
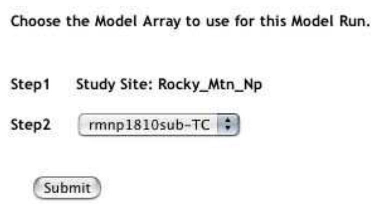
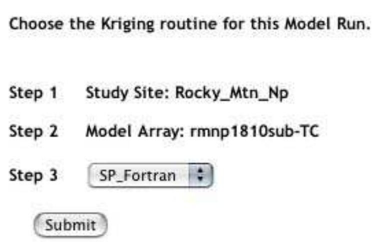
Graphical User Interface	**Back-end processes**
	1. User enters account name and password to enter the system. 2. User is authenticated in lookup table and enters the new system. OR User is not authenticated and the jsp displays an error message "Your Account name or password is invalid."
	1. A list of Study Sites from the DB (or from file system) that are ready to be submitted as a model run are displayed. 2. User selects study site and submits selection.
	1. A list of Model Arrays (MA) from DB (or from file system) is displayed. 2. User selects model array and submits selection.
	1. A list of Kriging routines from DB (or from file system)is displayed. 2. User selects Kriging routine and submits selection.

Table 6 Continued on next page

Table 6. Graphical User Interface

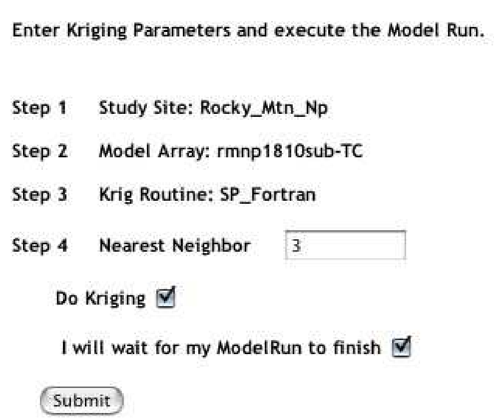
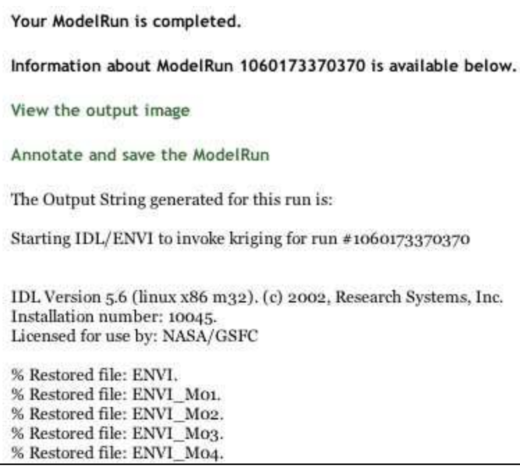
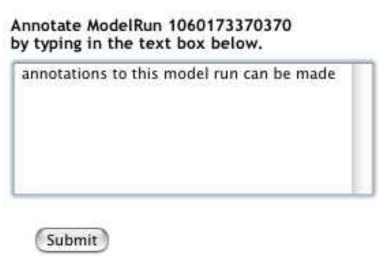
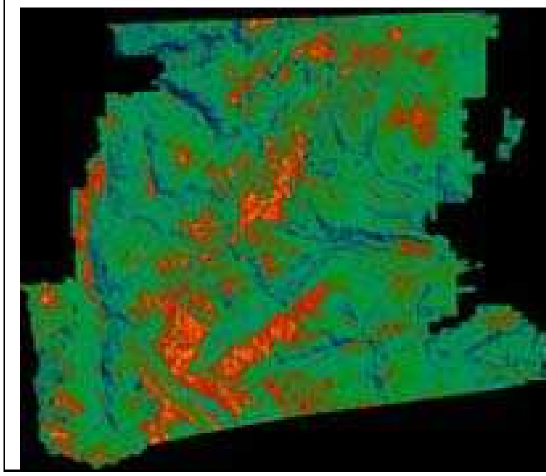
Table 6 Continued from previous page	
Graphical User Interface	**Back-end processes**
	1. Use enters number of nearest neighbors to be calculated. 2. User selects whether to incorporate Kriging. 3. User selects whether to wait for the Model-Run to be completed. 4. User selects Kriging routine and submits selection and submits selections.
	1. The metadata for the run is displayed (possibly requiring a call to the DB to get it). User can select to “View the Output Image” or “Annotate and save the ModelRun.”
	1. When user selects to Annotate and save the ModelRun, a text box is displayed for annotations. 2. User annotates and submits text.
	The user comments are saved to a field within the metadata record.
Table 6 Continued on next page	

Table 6 Continued from previous page	
Graphical User Interface	**Back-end processes**
	When user selects to View the output image, the image file from the ModelRun is drawn on the screen.

5 Data Interfaces

The data interfaces employed by the subsystem are detailed in the following data dictionary and ER diagram.

5.1 Data Dictionary

The following tables enumerate database tables and attributes, which are used by the BP web application system. The purpose of this Data Dictionary is to describe

1. the function of each table,
2. descriptions of attributes within a table including their modifiers and constraints, and
3. the relationships and dependences of tables.

5.1.1 krigparams

A table used to support configurable kriging parameters. Krig routines may potentially use many different parameters. This table contains data that specifies possible parameters used by kriging algorithms. It is used by the web application to display parameter setting to the user for their input.

Column Name	Type	Modifiers	Description
name	varchar	not null	The name for this parameter.
type	varchar	not null	The IDL data type of the parameter.
description	text		An optional meaningful description of this parameter.

Table 7. KRIGPARAMS

5.1.2 krigroutine

Table used to support definitions for multiple kriging algorithms/routines. Users may choose one of the kriging algorithms in this table for each model run.

Column Name	Type	Modifiers	Description
name	varchar	not null	The name of the krig routine.
krigparamsname	varchar	not null	A parameter used by the kriging algorithm. The foreign key for a row in the krigparams table. Currently we only support one kriging parameter.
uri	varchar	not null	Currently not used.
description	text		An optional detailed description of this kriging algorithm.

Table 8. KRIGROUTINE

5.1.3 mergeddataset

A merged data set is a pool of data from which a modelarray is extracted. Is the complete dataset for a study site.

Column Name	Type	Modifiers	Description
name	varchar	not null	The name of the data set.
studysitename	varchar	not null	The foreign key for a row in the study site table.
uri	varchar	not null	Currently not used.
variablesname	varchar	not null	Variable name that is contained in the data set. Currently there is only one variable name. This table must be changed to support many variable names.
description	varchar	not null	A detailed description of this data set.

Table 9. MERGEDDATASET

5.1.4 modelarray

Subset of the mergeddataset used to run against the kriging algorithm. Contains variables from the mergeddataset that may be predictors in the forecasting of scenarios.

Column Name	Type	Modifiers	Description
name	varchar	not null	The name of the model array.
mergeddataname	varchar	not null	Specifies which data set this model array was generated from. The foreign key for a row in the mergereddataset table.
url	varchar	not null	Currently not used.
predictorvar	varchar	not null	The variable name to use as a predictor. Currently we only support one. This table will have to change to support many.
description	varchar	not null	A detailed description.
version	varchar	not null	Each model array is versioned.

Table 10. MODELARRAY

5.1.5 modeloutput

Holds the console output from the model run and descriptors for the model run such as a timestamp and a locator for the image produced.

5.1.6 modelrun

Holds parameters for specifying a model run. A row will exist for each model run requested on the system.

Column Name	Type	Modifiers	Description
modelname	varchar	not null	The logical name of the modeloutput. The name is the same as the name of the modelrun. It is a foreign key for a row in the modelrun table.
outputinfo	Text	not null	Text from the modelrun output.
timegenerated	varchar	not null	Timestamp of the modelrun.
imageurl	varchar	not null	Created from the modelrun output image file name.

Table 11. MODELOUTPUT

Column Name	Type	Modifiers	Description
name	varchar	not null, unique	A unique logical name for a particular model run.
modelarrayname	varchar	not null	The name of the model array used in this model run. A foreign key to the modelarray table.
krigname	varchar	not null	Specifies the krig routine used in this model run. Foreign key to krigroutine table.
description	text		An optional text description for a particular model run.
hasrun	boolean		True if the model run has been executed and finished.

Table 12. MODELRUN

5.1.7 studysite

This table is an index of study sites for which we have a data set.

Column Name	Type	Modifiers	Description
name	varchar	not null	The name of the study site. Currently this is the logical name used by the kriging algorithm.
uri	varchar	not null	Image identification reference tool.
description	text		A detailed description of the site.

Table 13. STUDYSITE

5.1.8 useraccount

The useraccount table holds user information. Currently the passwords are not encrypted.

Column Name	Type	Modifiers	Description
accountname	varchar	not null	The username. This should be the user's email address for notification to work.
password	varchar	not null	The plain-text password for the user. We may want to encrypt it in the future.

Table 14. USERACCOUNT

5.2 Entity Relationship (ER) Diagram

Figure 12 illustrates the relationship between database tables (entities). In the diagram, rectangles represent entities and diamonds represents the relationship between entities. The diagram should be read from left to right and top to bottom.

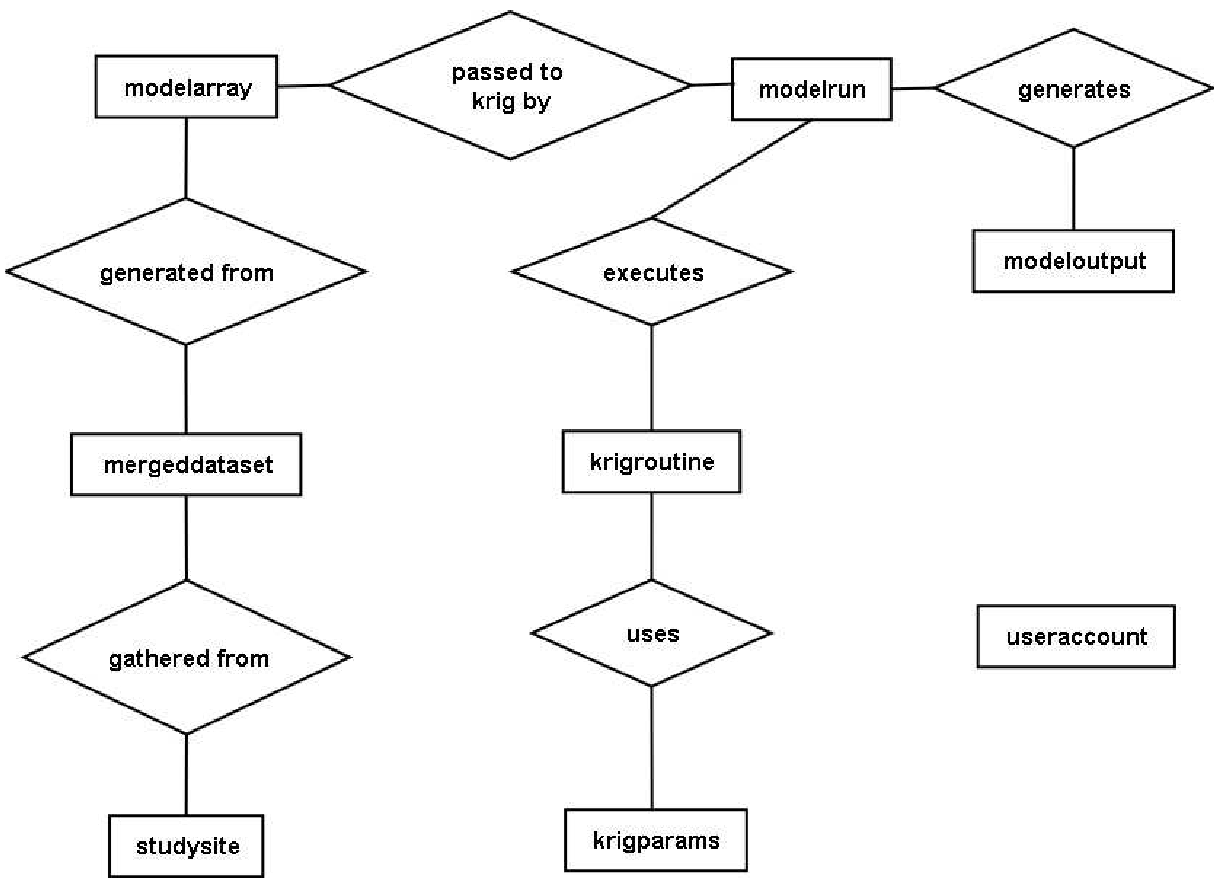


Figure 12. ER Diagram

A Glossary

BP Biotic Prediction project
CT Computational Technologies project
CONOP Concept of Operations
COTS Commercial Off The Shelf
CSU Colorado State University
CVS Concurrent Version Software
DAAC Distributed Active Archive Center
DAO Data Access Object
DTO Data Transfer Object
ENVI Environment for Visualizing Images
ESTO Earth Science Technology Office
FTP File Transfer Protocol
GSFC Goddard Space Flight Center
GUI Graphical User Interface
HTML Hyper-Text Markup Language
HTTP Hyper-Text Transport Protocol
IDE Integrated Development Environment
IDL Interactive Data Language
ISFS Invasive Species Forecasting System
J2EE Java 2 Platform, Enterprise Edition
J2SE Java 2 Platform, Standard Edition
JDBC Java Database Connectivity
JNDI Java Naming and Directory Interface
JOX Java Obljects in XML
LAN Local Area Network
LINUX Open source Unix-type operating system
MVC Model View Controller
NBII National Biological Information Infrastructure
NREL Natural Resources Ecology Laboratory
RDBMS Relational Database Management System
SCP Secure Copy
SDD Software Design Document
SSH2 Secure Shell
UML Unified Modeling Language
URL Uniform Resource Locator
USGS United States Geological Survey
XML Extensible Markup Language

B Structure Charts & Object-Oriented Design

B.1 Introduction

ISFS incorporates JavaServer Pages on the back-end to make database calls and activate the application. Development of the system is done using a Java framework known as Jakarta Struts.

The core of the Struts framework is a flexible control layer based on standard technologies like Java Servlets, JavaBeans, ResourceBundles, and Extensible Markup Language (XML), as well as various Jakarta Commons packages. Struts encourage application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller (MVC) design paradigm.

Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. For the Model, Struts can interact with any standard data access technology, including Enterprise Java Beans, JDBC, and Object Relational Bridge. For the View, Struts works well with JavaServer Pages, including JSTL and JSF, as well as Velocity Templates, XSLT, and other presentation systems.

The Struts framework provides the invisible underpinnings that a professional web application needs to survive. Struts help to create an extensible development environment for the application, based on published standards and proven design patterns.

Struts is part of the Apache Jakarta Project, sponsored by the Apache Software Foundation. Additional information is available from the official Struts home page http://jakarta.apache.org/struts.

The following sections break down the Struts construction of the ISFS application in terms of Model Run Design, User Accounts Design, ISFS Common Design, Package and Class Hierarchies. These models and diagrams form the basis for ISFS application development.

B.2 Model Run Design

B.2.1 Overview and Responsibilities

The ISFS modelrun package contains classes that encapsulate the ModelRun concepts, and implement the execution, and editing of ModelRun objects. Like all of the other packages it is broken up into four sub-packages, app, bus, db and valueObj. The main interface for ModelRun activity is the ModelRunService class, which gets constructed and placed into the session when a user logs into the system. From there, the ModelRunService is used by the app classes for all of the ModelRun assembly, execution and editing processes.

B.2.2 Class Descriptions

B.2.2.1 bus package

<u>ModelRunService:</u> class used to access all information needed to alter,access and execute ModelRuns. Serves as the interface for the application classes.

<u>SecureCopyThread:</u> Implementation of SCP used to securely copy files to and from the ISFS ModelRun kriging server.

<u>ModelRunController:</u> controller class that serves as the mechanism for submitting ModelRun jobs. Currently, this class holds a queue of jobs and executes each job via SSH in the order that the jobs were submitted. In the future, the plan for this class would be to communicate with the ModelRunServer via RMI for job submission and the ModelRunServer would handle each individual job.

ModelRunServer: Not Implemented, described briefly above.

MyUserInfo: Class needed by the SSH and SCP classes to give user information.

B.2.2.2 app package

BuildModelRunAction: implementation of the Struts DispatchAction class which contains methods for the sequence of assembling a ModelRun and having it execute.

BuildModelRunActionForm: implementation of the Struts ActionForm class which is populated across a multiple pages, once the form is completed its values are used to execute a ModelRun.

AnnotateModelActionForm: implementation of the Struts ActionForm class, which is filled in during the annotation of a ModelRun. Its data is eventually stored in the database as an annotation to the modelrun.

EditModelRunAction: implementation of the Struts DispatchAction class that contains methods for editing and viewing ModelRuns.

B.2.2.3 valueObj package

ModelRun: the encapsulation of a ModelRun, which contains all of the assembly information such as study site, modelarray, etc., that is needed to execute a model run . ModelRuns are created before they are run so some ModelRuns have no ModelOutput yet.

ModelOutput: encapsulated results of a ModelRun.

MergedDataSet: encapsulation of a mergeddataset file.

StudySite: describes a site from which data and images were drawn.

KrigParams: parameters class encapsulating the parameters of a KrigRoutine.

ModelArray: encapsulates the model array data file used as the data in a ModelRun.

ModelRunAssemblyInfo: convenient bean used to store all of the possible combinations of data that could be used to assemble a ModelRun. This bean is filled when a user starts to assemble a ModelRun and is stored in the session along with the ModelRunService, so that the system does not have to repeatedly access the database to get this information.

KrigRoutine: representation of a KrigRoutine which contains attributes to describe itself and its possible parameters along with the location from which it can be used.

StatusInfo: bean placed in the session when a ModelRun is submitted, which keeps an attribute that is the percentage of completion for the ModelRun submitted in this session.

Note: db package **contains DAO objects for all of the classes in the valueObj package**.

B.2.3 Diagrams

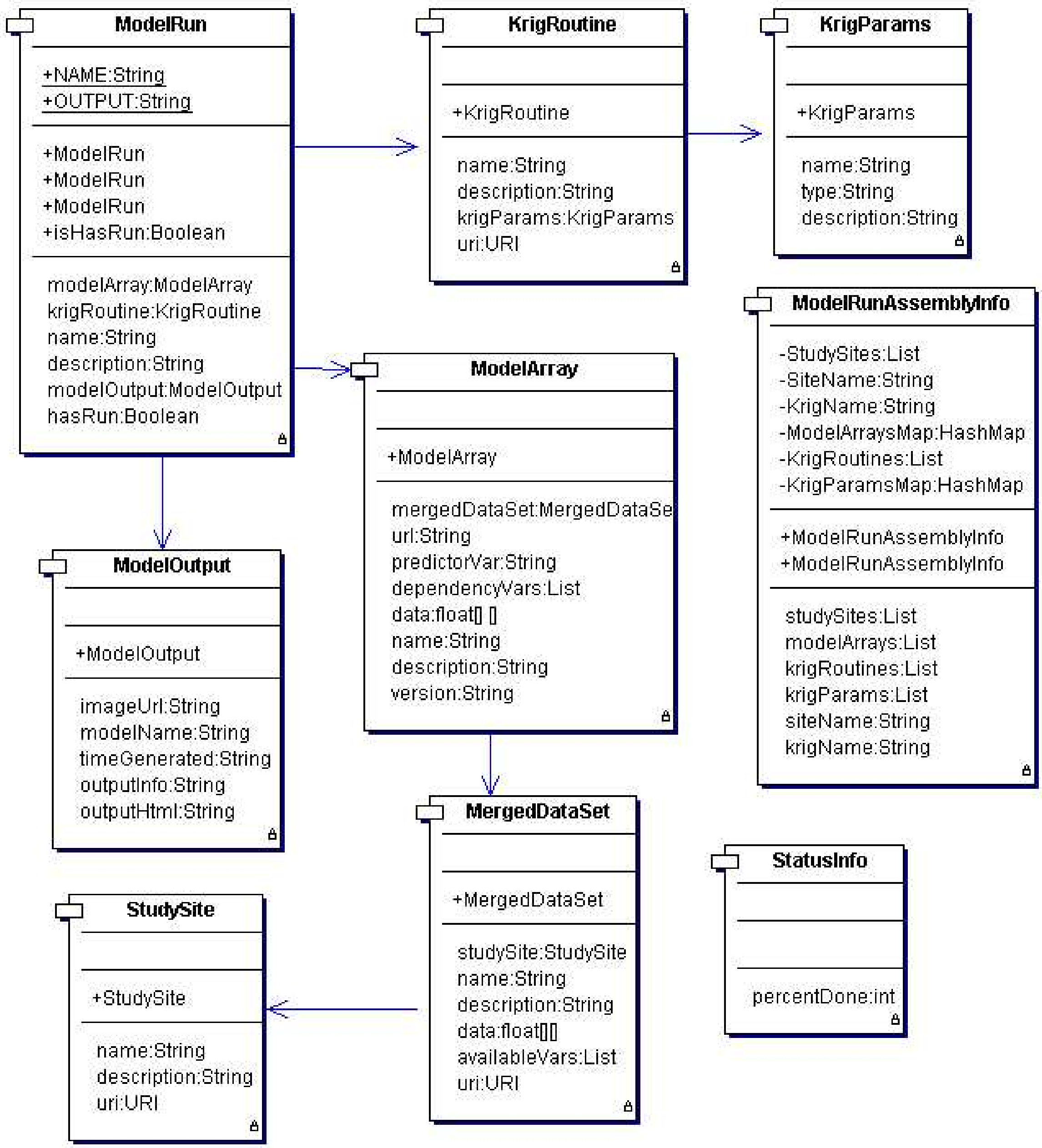


Figure 13. modelrun\valueObj class diagram

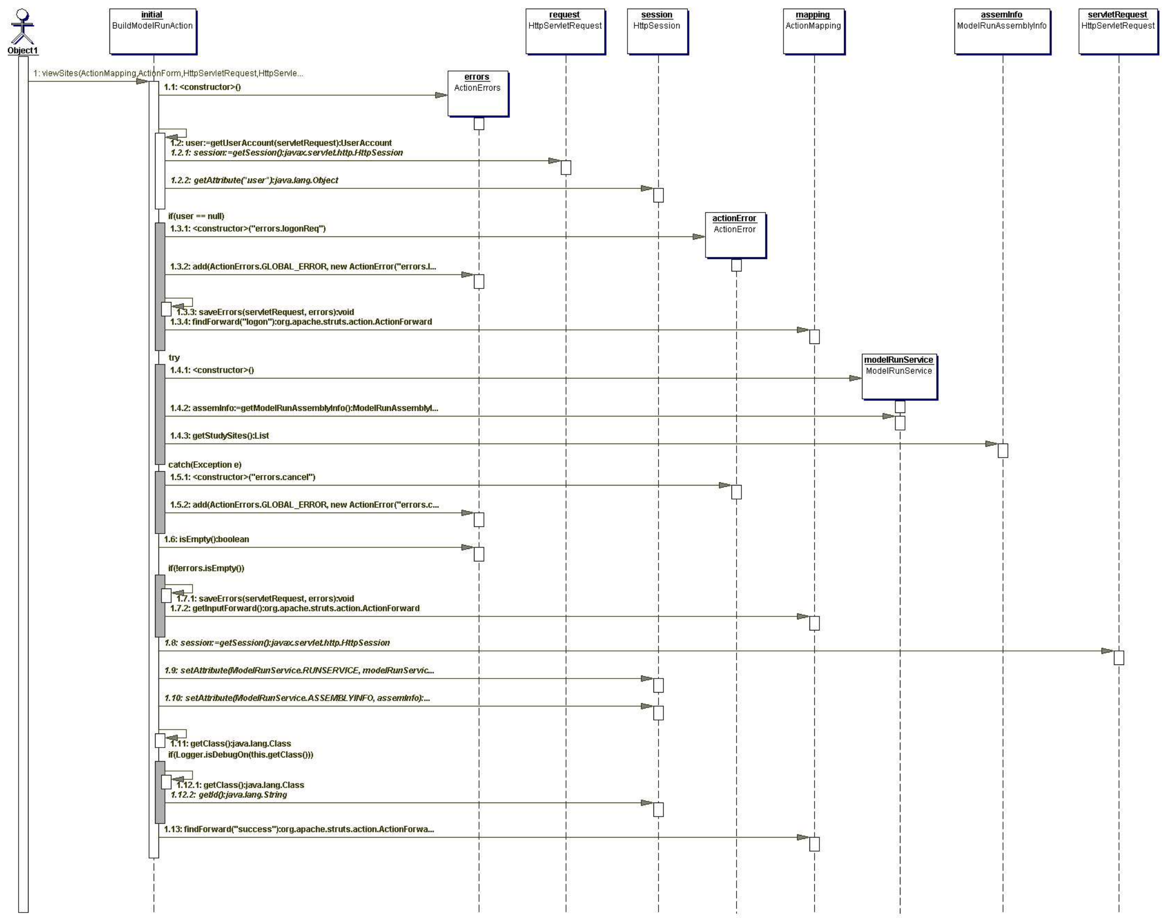


Figure 14. The sequence diagram for the beginning of ModelRun assembly as the user is presented with a list of study sites.

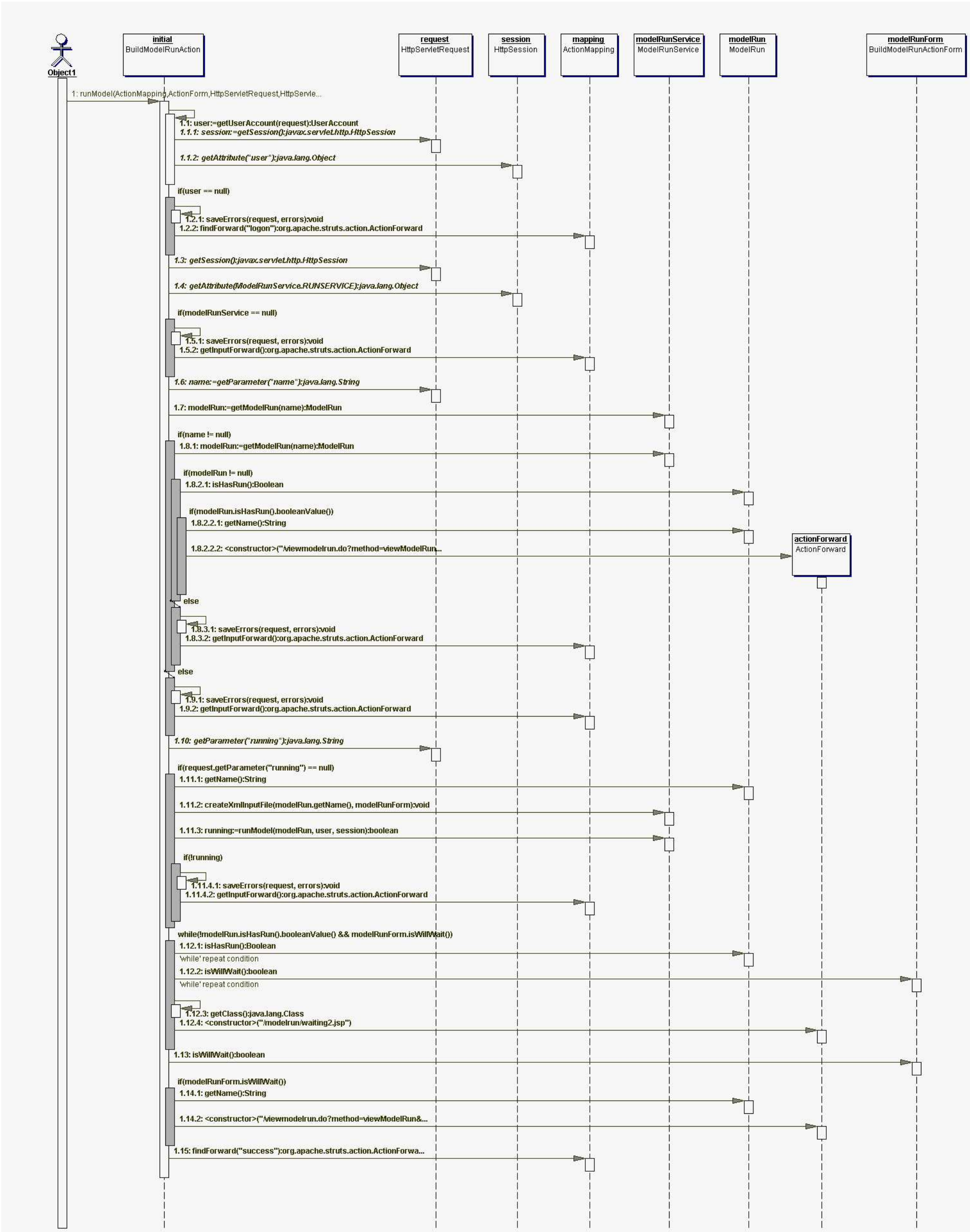


Figure 15. Sequence diagram displaying the submission of a ModelRun job by the user.

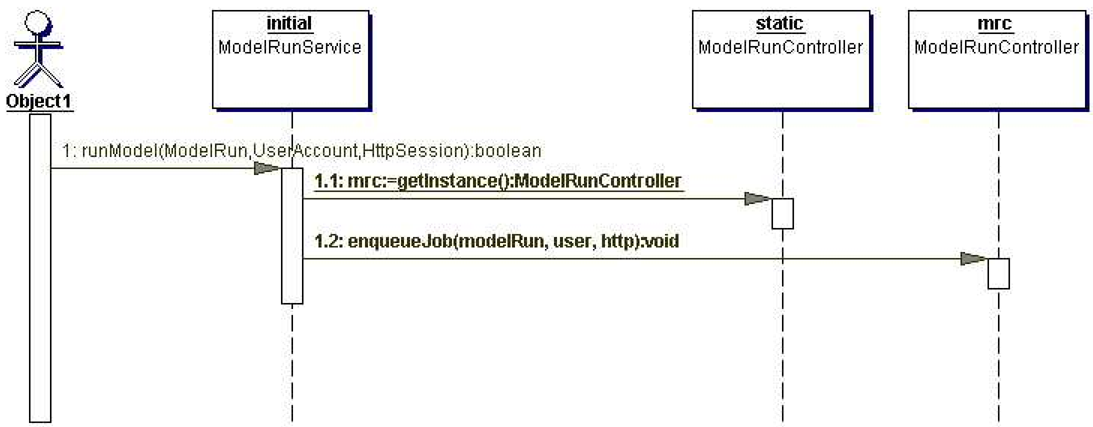


Figure 16. Sequence diagram displaying the runModel method of the BuildModelRunAction class, which places the job into the ModelRunControllers job queue.

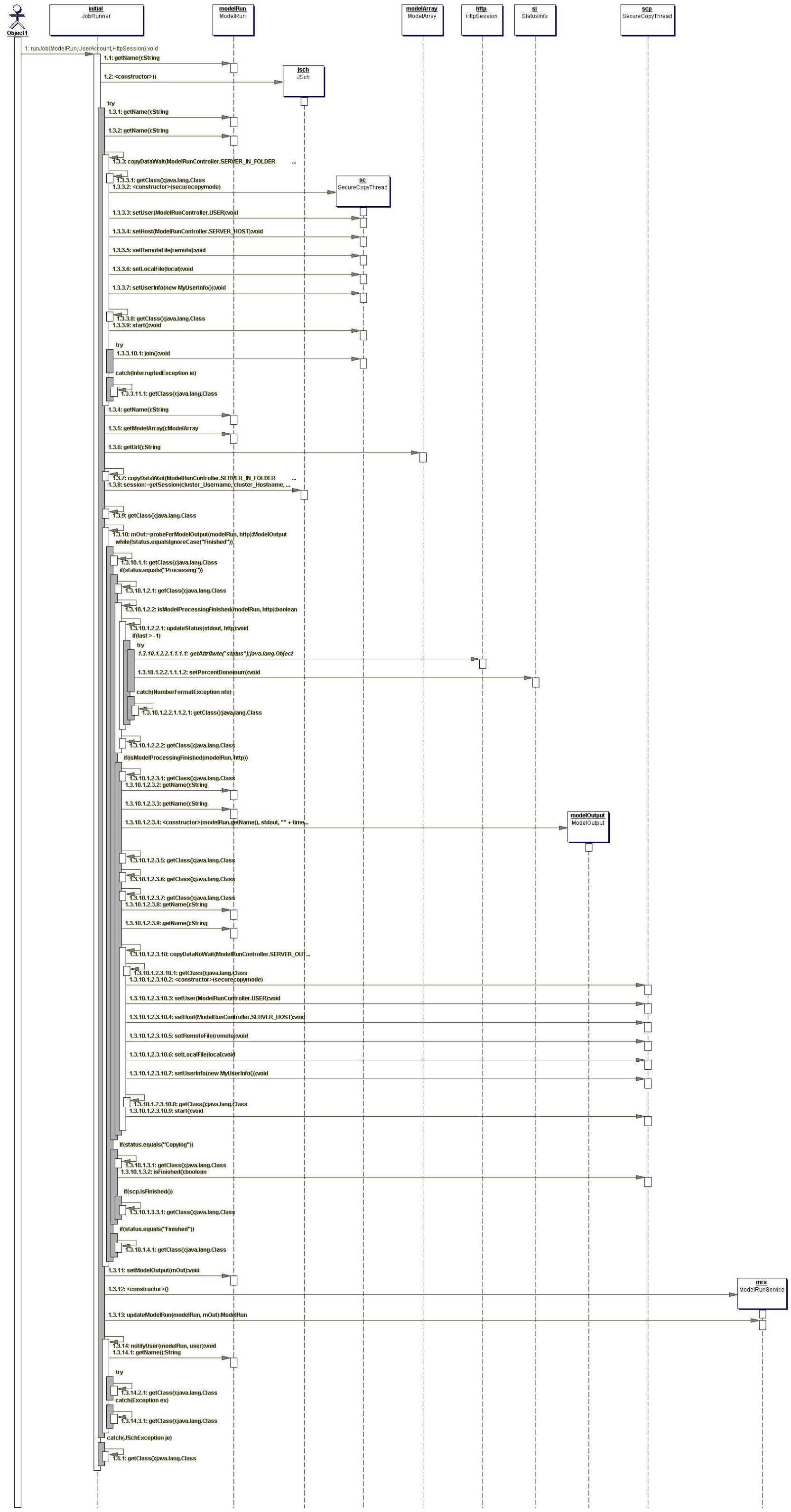


Figure 17. Sequence diagram displays execution of a ModelRun by the ModelRunController and its internal JobRunner class.

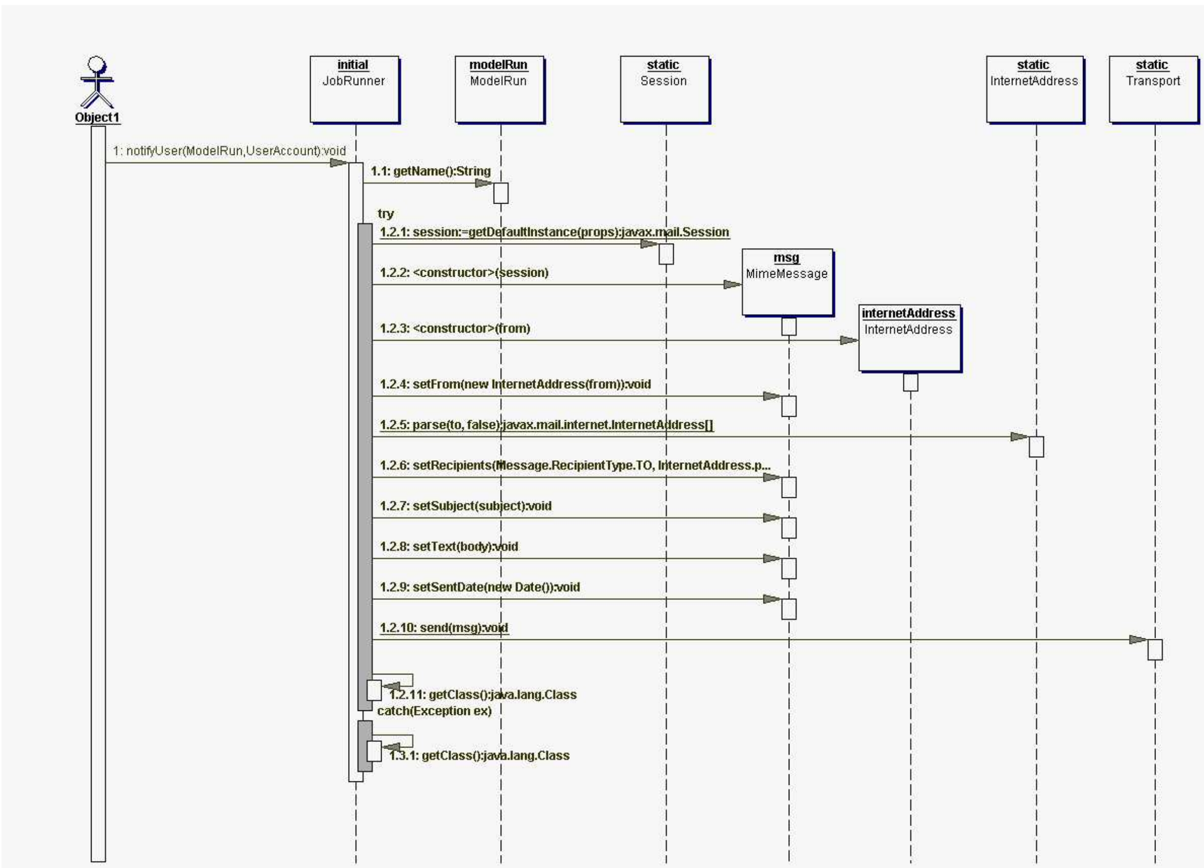


Figure 18. Sequence diagram showing how the ModelRunController notifies the user when a ModelRun is complete.

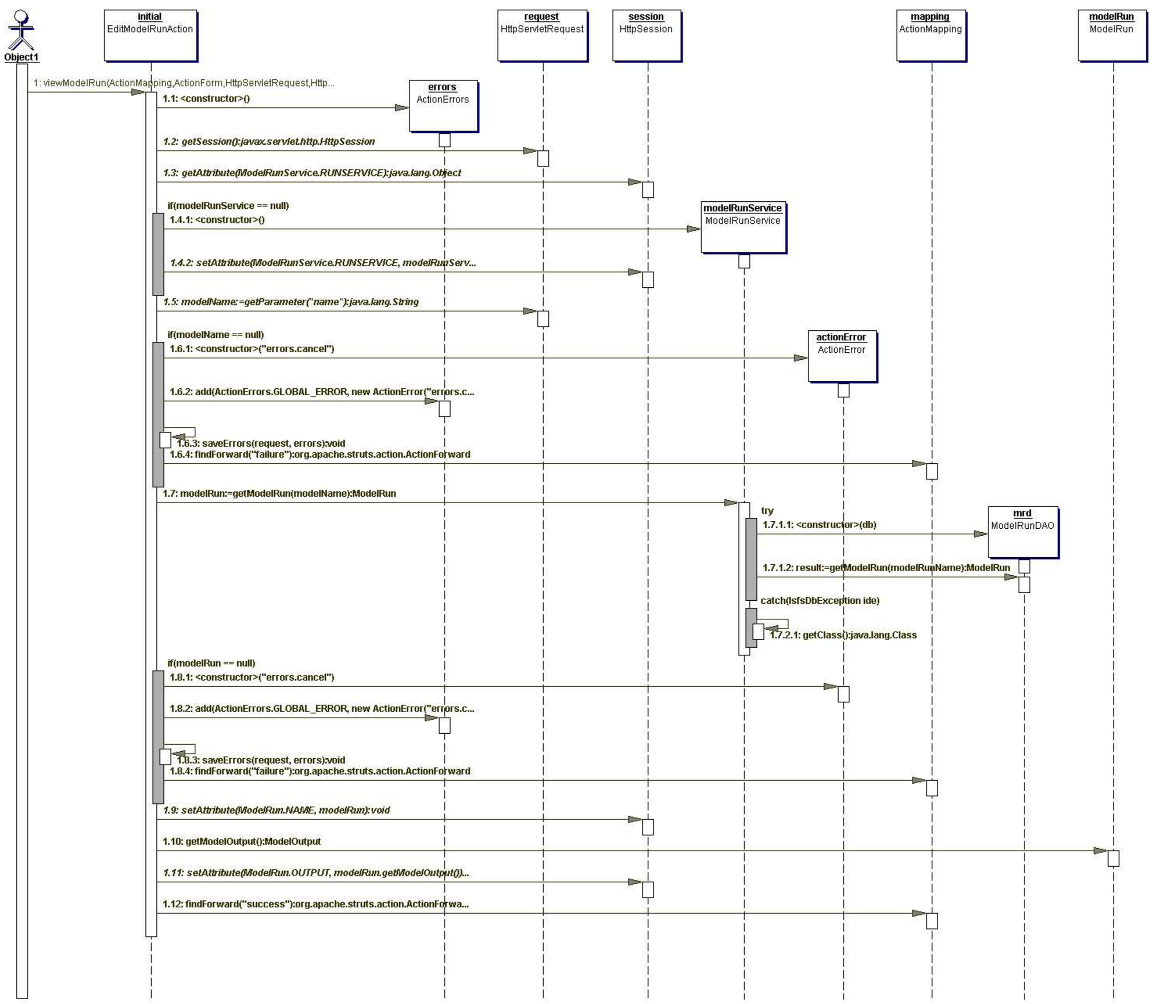


Figure 19. Sequence diagram showing what happens when a user views a completed ModelRun.

B.3 User Accounts Design

B.3.1 Overview and Responsibilites

The ISFS ModelRun tool is intended to be used by authorized users only. Currently there is only one type of a user and he may execute, view and annotate ModelRuns. In future releases, this must be expanded to add multiple levels of users and user capabilities that will include roles for administrators, ModelArray/data creators and more. Security measures for user passwords and authorization levels must also be implemented.

The UserAccountsManager should eventually have the following responsibilities:

- provide an interface for the user access to the system;
- provide information about individual users;
- encrypt and decrypt user passwords;
- update user information; and
- create and remove users for the system

B.3.2 Class Descriptions

bus package

UserAccountsManager: the primary class with which the Presentation Layer interfaces. The UserAccountsManager provides access to all user operations.

valueObj package

UserAccount: encapsulates ISFS user information. This object is created when a user logs onto the system and is placed in the users http session.

app package

LogonAction: the struts action used to logon users.

LogonActionForm: the Struts ActionForm for the logon process.

dbPackage

UserAccountDAO: the database access object for user accounts. This class knows how access all of the user information stored in the database and how to construct a UserAccount form the stored data.

B.3.3 Diagrams

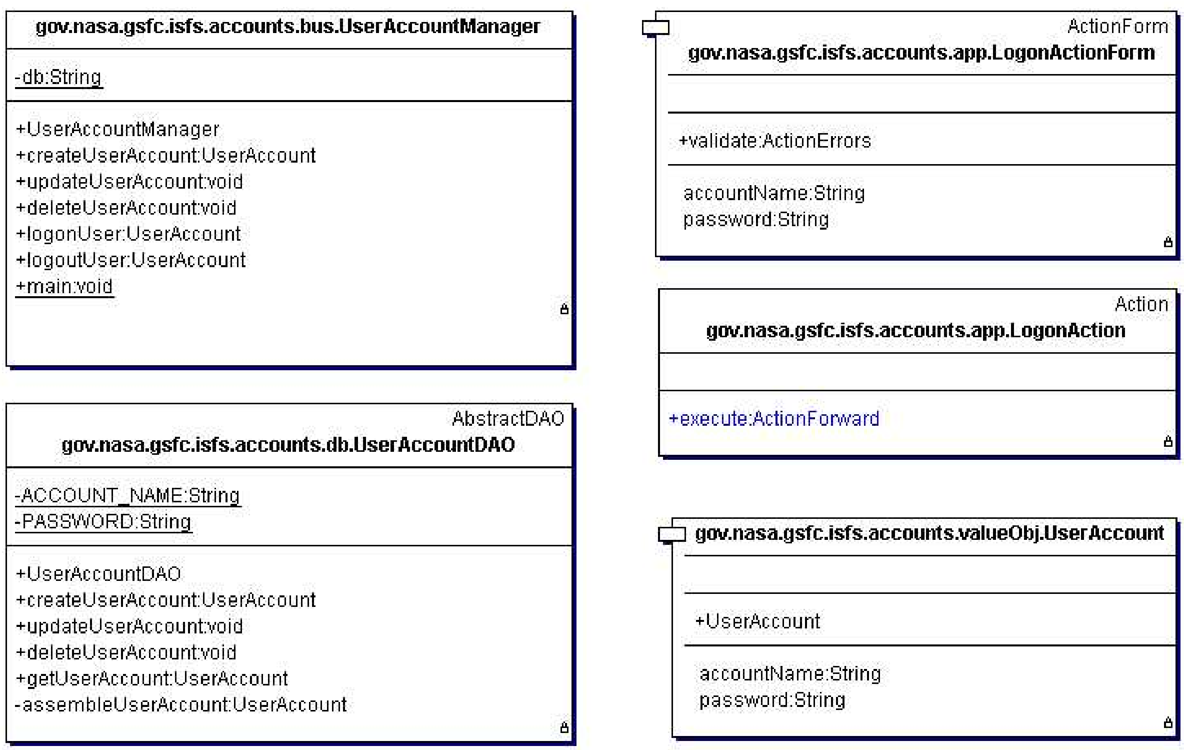


Figure 20. Accounts Package (Class Diagram)

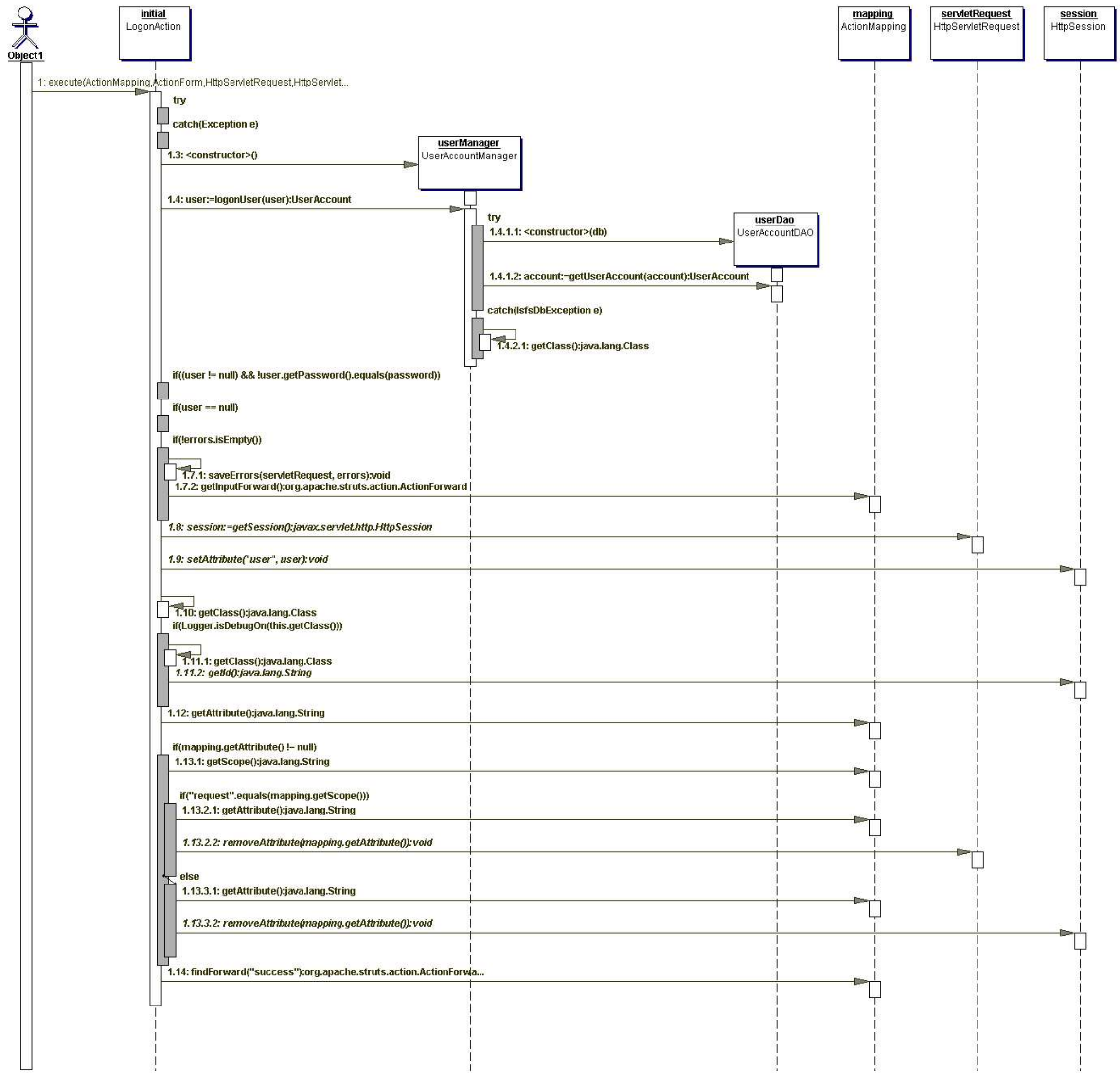


Figure 21. Logon to System (Sequence Diagram)

B.4 ISFS Common Design

B.4.1 Overview and Responsibilities

The ISFS common package contains all of the shared database and utilities classes needed for the ISFS ModelRun tool. This is the package where we place all of the utilities and classes that do not fit well within other packages.

B.4.2 Class Descriptions

AbstractDAO: this class handles all of the generic DAO JDBC functionality. It is intended to be extended with specific functionality for the object that needs DAO support.

ISFSDAO: this class is intended to have more specific DAO functionality for the ISFS project. Many of the ISFS DAO objects extend this class.

Transaction: encapsulates a database transaction and is used within the AbstractDAO class.

ISFSDbException: ISFS Exception wrapper class around the JDBC database exception.

Logger: ISFS wrapper class around the Log4j logging.

B.4.3 Diagrams

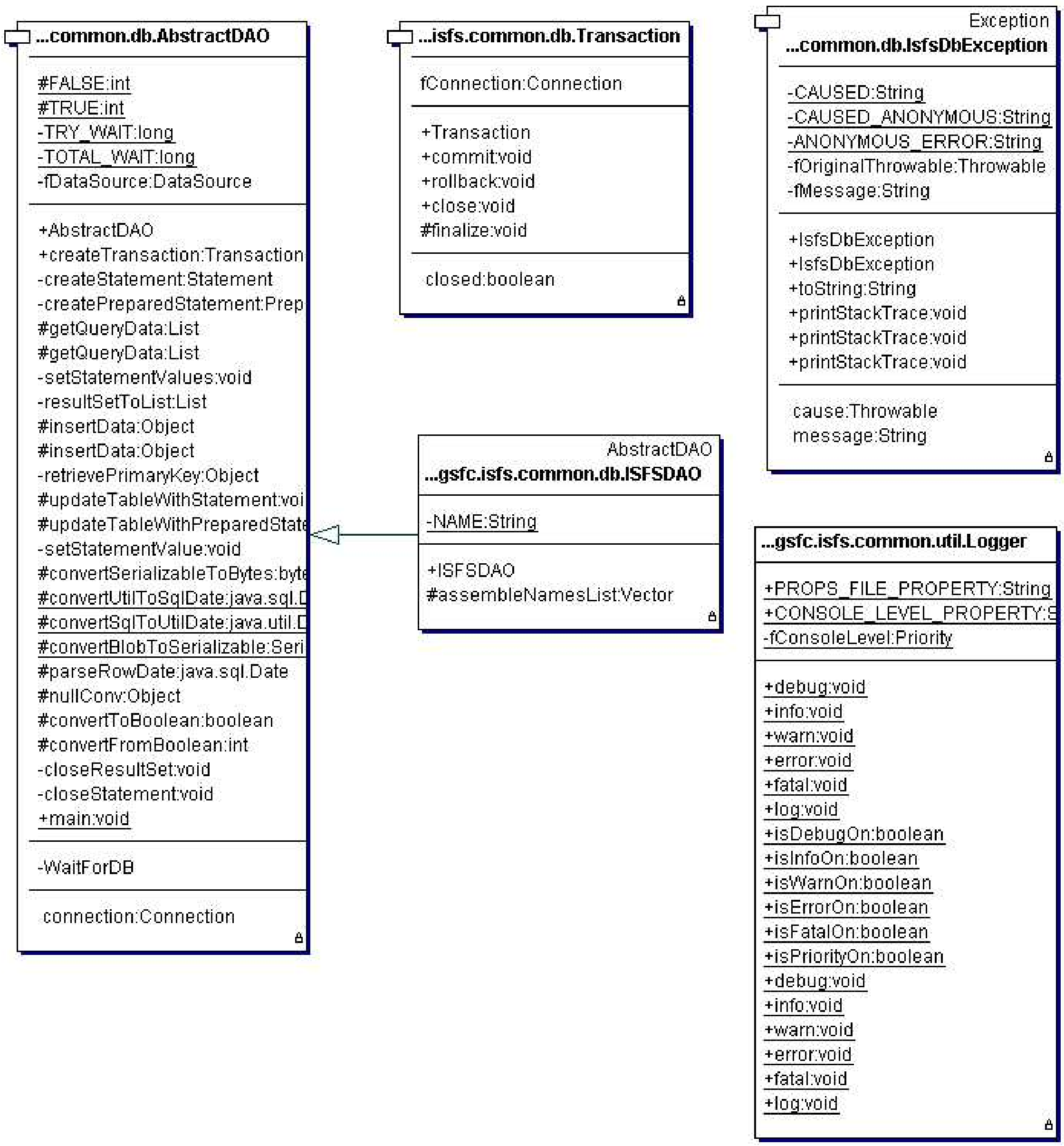


Figure 22. Common package (Class Diagram).

B.5 Hierarchy For All Packages

Package Hierarchies:

<default>, design, design.doc-files, gov, gov.doc-files, gov.nasa, gov.nasa.doc-files, gov.nasa.gsfc, gov.nasa.gsfc.doc-files, gov.nasa.gsfc.isfs, gov.nasa.gsfc.isfs.accounts, gov.nasa.gsfc.isfs.accounts.app, gov.nasa.gsfc.isfs.accounts.app.class-use, gov.nasa.gsfc.isfs.accounts.app.doc-files, gov.nasa.gsfc.isfs.accounts.bus, gov.nasa.gsfc.isfs.accounts.bus.class-use, gov.nasa.gsfc.isfs.accounts.bus.doc-files, gov.nasa.gsfc.isfs.accounts.db, gov.nasa.gsfc.isfs.accounts.db.class-use, gov.nasa.gsfc.isfs.accounts.db.doc-files, gov.nasa.gsfc.isfs.accounts.doc-files, gov.nasa.gsfc.isfs.accounts.valueObj, gov.nasa.gsfc.isfs.accounts.valueObj.class-use, gov.nasa.gsfc.isfs.accounts.valueObj.doc-files, gov.nasa.gsfc.isfs.common, gov.nasa.gsfc.isfs.common.db, gov.nasa.gsfc.isfs.common.db.class-use, gov.nasa.gsfc.isfs.common.db.doc-files, gov.nasa.gsfc.isfs.common.doc-files, gov.nasa.gsfc.isfs.common.util, gov.nasa.gsfc.isfs.common.util.class-use, gov.nasa.gsfc.isfs.common.util.doc-files, gov.nasa.gsfc.isfs.doc-files, gov.nasa.gsfc.isfs.modelrun, gov.nasa.gsfc.isfs.modelrun.app, gov.nasa.gsfc.isfs.modelrun.app.class-use, gov.nasa.gsfc.isfs.modelrun.app.doc-files, gov.nasa.gsfc.isfs.modelrun.bus, gov.nasa.gsfc.isfs.modelrun.bus.class-use, gov.nasa.gsfc.isfs.modelrun.bus.doc-files, gov.nasa.gsfc.isfs.modelrun.db, gov.nasa.gsfc.isfs.modelrun.db.class-use, gov.nasa.gsfc.isfs.modelrun.db.doc-files, gov.nasa.gsfc.isfs.modelrun.doc-files, gov.nasa.gsfc.isfs.modelrun.valueObj, gov.nasa.gsfc.isfs.modelrun.valueObj.class-use, gov.nasa.gsfc.isfs.modelrun.valueObj.doc-files, gov.nasa.www, gov.nasa.www.doc-files, html_docs

B.6 Class Hierarchy

- class gov.nasa.gsfc.isfs.common.db. AbstractDAO
  - class gov.nasa.gsfc.isfs.common.db. ISFSDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. KrigInfoDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. MergedDataSetDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. ModelArrayDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. ModelOutputDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. ModelRunDAO
    - class gov.nasa.gsfc.isfs.modelrun.db. StudySiteDAO
  - class gov.nasa.gsfc.isfs.accounts.db. UserAccountDAO
- class gov.nasa.gsfc.isfs.modelrun.valueObj. KrigParams
- class gov.nasa.gsfc.isfs.modelrun.valueObj. KrigRoutine
- class gov.nasa.gsfc.isfs.common.util. Logger
- class gov.nasa.gsfc.isfs.modelrun.valueObj. MergedDataSet
- class gov.nasa.gsfc.isfs.modelrun.valueObj. ModelArray
- class gov.nasa.gsfc.isfs.modelrun.valueObj. ModelOutput
- class gov.nasa.gsfc.isfs.modelrun.valueObj. ModelRun
- class gov.nasa.gsfc.isfs.modelrun.valueObj. ModelRunAssemblyInfo
- class gov.nasa.gsfc.isfs.modelrun.bus. ModelRunController
- class gov.nasa.gsfc.isfs.modelrun.bus. ModelRunServer
- class gov.nasa.gsfc.isfs.modelrun.bus. ModelRunService
- class gov.nasa.gsfc.isfs.modelrun.bus. MyUserInfo (implements UserInfo)
- class java.lang.Object
  - class org.apache.struts.action.Action
    - class org.apache.struts.actions.DispatchAction
      - class gov.nasa.gsfc.isfs.modelrun.app. BuildModelRunAction
      - class gov.nasa.gsfc.isfs.modelrun.app. EditModelRunAction
    - class gov.nasa.gsfc.isfs.accounts.app. LogonAction
  - class org.apache.struts.action.ActionForm
    - class gov.nasa.gsfc.isfs.modelrun.app. AnnotateModelActionForm
    - class gov.nasa.gsfc.isfs.modelrun.app. BuildModelRunActionForm
    - class gov.nasa.gsfc.isfs.accounts.app. LogonActionForm
  - class java.lang.Thread
    - class gov.nasa.gsfc.isfs.modelrun.bus. SecureCopyThread
  - class java.lang.Throwable
    - class java.lang.Exception
      - class gov.nasa.gsfc.isfs.common.db. IsfsDbException
- class gov.nasa.gsfc.isfs.modelrun.valueObj. StatusInfo
- class gov.nasa.gsfc.isfs.modelrun.valueObj. StudySite
- class gov.nasa.gsfc.isfs.common.db. Transaction
- class gov.nasa.gsfc.isfs.accounts.valueObj. UserAccount
- class gov.nasa.gsfc.isfs.accounts.bus. UserAccountManager

C ISFS Developer Setup/Deployment Instructions

C.1 Tools and Environment Setup

C.1.1 WinCVS

C.1.1.1 Installation and Configuration

The ISFS project uses CVS for source control. You must install and setup a CVS client in order to access CVS. For Win32, MacOSX and Unix/Linux systems CVSGUI is available at the following location http://wincvs.org/
The CVS repository is located on the server carbon.sesda.com under the folder `/export/home/cvs`. You will need an account and a pserver password set up for yourself on carbon, contact Dave Kendig who is the admin on carbon.sesda.com.
Once you are set up on carbon and you have a CVS client you can link up to CVS by setting your `CVSROOT=pserver:account_name@carbon.sesda.com:/export/home/cvs` where account_name is your account name on carbon.

C.1.1.2 ISFS CVS Details

- You will need to create an ISFS directory on your machine. This is where all of your ISFS work will reside. For example: `C:\ISFS` or `C:\Data\ISFS`
- In CVS there is one module that holds everything for ISFS. The module is named “BP”. To get the module from CVS do the following:
  1. Create → Checkout Module...
  2. On the Checkout Setting tab, enter “BP” for the module name. Select the root ISFS directory as the directory to checkout the module to for example, `C:\Data\ISFS`

C.1.2 Java

J2SE 1.4 is required and is available at the Java website: http://java.sun.com/

C.1.3 Ant

ISFS uses Ant as a build tool. To get ANT you can download from: http://ant.apache.org/
The Ant build file for ISFS is located in the /BP/bin folder. Read this file to see what targets exits. In order to run Ant you must have the Environment Variables “JAVA_HOME” set equal to the path to Java 1.4 and “ISFS_HOME” set to equal the `path_to_tomcat_installation\webapps\isfs`.
The isfsant.bat file is located in the /BP/bin folder and should be edited with your path to the Ant executable, or you can use an environment variable, your choice. The batch file takes the target options and passes it to the executable, for example “> isfsant compile” will compile the ISFS project using Ant.

C.1.4 Tomcat

Tomcat provides the Servlet and JSP engine that ISFS uses to run the Portal. To get Tomcat you can download it from http://jakarta.apache.org/tomcat/. Currently we are using Tomcat 4.1.18, but more current version should work fine. It is also possible to use any other Servlet and JSP engine of your choice but currently we use Tomcat. The web.xml file for ISFS is located in the /BP/config folder in case you need to alter that file.

C.1.5 TogetherJ

The ISFS project has a copy of TogtherJ which is used as the UML/IDE tool. You are not required to use TogetherJ as your chosen tool, however to take advantage of the engineering information to date, you should use TogetherJ. The TogetherJ project file, ISFS.tpr, is located in the BP folder of the project.

C.1.6 Database

The database used for ISFS is a Postgres database that is running on port 5432 on the carbon.sesda.com server. The name of the database is isfs. The schema file is located in the /BP/bin folder and is named isfs.sql. The poolman config file needs editing if you switch databases. This file is described in more detail in the ”Configuration files” section of this document. To login to the database on carbon, login to carbon and issue the following command: `/usr/local/pgsql/bin/psql -U postgres isfs` which will log you in to the “isfs” database as the user “postgres”. The Postgres slash commands come in handy for describing info about the database and it’s tables.

C.1.7 Struts

ISFS is using the Jakarta Struts web application framework to help make design and implementation of ISFS a lot easier, details about Struts can be found at http://jakarta.apache.org/struts/. The Struts config file is located in the /BP/config folder and is described in more detail in the “Configuration files” section of this document. If you have not used Struts before it is recommended that you familiarize yourself with Struts and it’s concepts before altering ISFS code. The O’reilly book Programming Jakarta Struts is a good resource to have and the Struts website has plenty of documentation.

C.1.8 Configuration Files

During the development process, it is possible to use your own local database and tomcat servers so as not to use the production versions for testing your development. Some changes would need to be made to config files in order to do this.

- Poolman Changes for alternative database

  ISFS uses poolman to do connection pooling. Poolman’s configuration is based on a poolman.xml file. The xml file defines how to connect to each database that ISFS uses in addition to a bunch of parameters that define how the pool should function.

  The file is located at `\BP\config\poolman.xml`

  You need to alter the entry for the `<!-- Standard JDBC Driver info -->` section of the datasource element so that it points to your database.

  Change `<dbname>` to the name of the database on your local db server.

  Change: `<url>jdbc postgresql://carbon.sesda.com:5432/isfs</url>`
  to: `<url>jdbc:postgresql://machine_name:port/datbase_name</url>`

  Change the user name and password appropriately.

If you are using a database other than Postgres make sure that you have the appropriate driver in the /BP/lib folder.

- isfs.props

  The isfs.props file holds properties used by the core of ISFS to figure out locations of servers needed during the running of ISFS models. If you are developing locally(i.e. your own local version of Tomcat or other changes) you must change the values of these properties.

  For example, if you are running a local Tomcat server, then you should change
  `WEBSERVER_HOME=full_path_to_tomcat_installation`
  `WEB_HOST=http://localhost:8080`

  Description of props:

  `WEBSERVER_HOME` — full path to web server home
  `WEBSERVER_DATA_FOLDER` — path to folder on web server containing the modelarray data.
  `WEBSERVER_IMAGES_FOLDER` — path to folder on web server containing output images.
  `SCRIPT_LOCATION` — location on server host containing kriging script
  `SCRIPT_NAME` — name of the kriging script
  `USER` — user name for the sever
  `SERVER_HOST` — kriging server host
  `SERVER_IN_FOLDER` —where the input data should be put for the run
  `SERVER_OUT_FOLDER` — where the output data will be put after the run
  `WEB_HOST` —the url to the web application host
  `MAIL_FROM` —the from line in emailed modelrun jobs
  `MAIL_SERVER` —name of the server that sends modelrun emails

The following config files are independent of sever development location but are config files nonetheless.

- web.xml

  The web.xml file is used by the Servlet and JSP engine(Tomcat) to figure out servlet request mappings. The web.xml file used in ISFS has Struts specific information in it and should only be altered by a developer that is familiar with web.xml files. It is located in the /BP/config folder.

- struts-config.xml

  The struts-config.xml file is where the struts application framework accesses all of the proper information specific to your struts application. Actions, Exceptions, Forms and Forward definitions are some of the elements in this file. Visit the http://jakarta.apache.org/struts/ site for more information. It is located in the /BP/config folder.

  An ISFS example: In the `..\isfs\accounts` package there is a file named LogonAction.java which contains an execute method. This class is an extension of the Struts Action class, the execute method gets called when the request forwards to the struts defined logon action which is defined in the struts-config.xml file like below:

```
<action-mappings>
    <action
        path="/logon"
```

```
            type="gov.nasa.gsfc.isfs.accounts.app.LogonAction"
            name="logonForm"
            scope="request"
            validate="true"
            input="/modelrun/logon.jsp">
        <forward name="success" path="/viewsites.do?method=viewSites"/>
        </action>
    </action-mappings>
```

- logger.props

  The logger.props file contains settings for the logger, such as how fine the logging information needs to be, i.e. show all debug info etc., and the location of the log file.

C.2 Running ISFS

C.2.1 Getting source

Use your CVS client (described above) to get the latest, and make sure that you have all of the tools described above.

C.2.2 Compiling

Run the script `\BP\bin\isfsant.bat`.
`isfsant.bat deploy` — will compile the Java code and move all of the isfs resources to the webserver for testing/deployment.

C.2.3 Running

- Make sure that the Database and Mail servers, to which you are connecting, are running.
- Run tomcat.

  Start your tomcat server by using tomcats start.bat or start.sh
- Open browser, and connect to ISFS at http://localhost:8088/isfs/modelrun/logon.jsp of course localhost:8088 should be server name and port number depending on your dev setup.

C.3 Deployment

Deploying the ISFS application is the same as setting it up for developing except that all of the defaults in the config files are set to work for the deployment machine Carbon.sesda.com, thus the user does not have to make any changes to the config files. If the deployment machine/s changes, then the files will need to be edited like above.
To deploy, follow the instructions above (taking into consideration the above comment) and once you have finished compiling using the “isfsant.bat deploy” command, you should then do the “isfsant.bat war” command to package up the application into a web archive and place it into the \webapps folder on your production machines version of Tomcat. Then restart Tomcat.